Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture

MPCFPE32B
Rev. 3, 9/2005

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

email:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
(800) 521-6274
480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064, Japan
0120 191014
+81 2666 8080
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate,
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
(800) 441-2447
303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: MPCFPE32B
Rev. 3, 9/2005

freescale™
semiconductor

Contents

Contents

Chapter 3
Operand Conventions

Contents

Chapter 4
Addressing Modes and Instruction Set Summary

Contents

Contents

Chapter 5
Cache Model and Memory Coherency

Contents

Chapter 6
Interrupts

Contents

Chapter 7
Memory Management

Contents

Contents

Contents

Contents

Contents

Figures

Figures

Figures

Figures

Figure Number	Title	Page Number

Tables

Tables

Tables

Tables

Tables

Tables

About This Book

The primary objective of this manual is to help programmers provide software that is compatible across a variety of implementations. Because the PowerPC™ architecture is designed to be flexible to support a broad range of processors, this book provides a general description of features that are common to these processors and indicates those features that are optional or that may be implemented differently in the design of each processor.

This revision of this book describes the 32-bit portion of the PowerPC architecture (Version 1.x) in detail.

Locate errata or updates for this document at www.freescale.com.

For designers working with a specific processor, this book should be used in conjunction with the user's manual for that processor.

This document distinguishes between the three levels, or programming environments, of the PowerPC architecture, which are as follows:

- User instruction set architecture (UISA)—The UISA defines the level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, memory conventions, and the memory and programming models seen by application programmers. U
- Virtual environment architecture (VEA)—The VEA, which is the smallest component of the PowerPC architecture, defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple processors or other devices can access external memory and defines aspects of the cache model and cache control instructions from a user-level perspective. VEA resources are particularly useful for optimizing memory accesses and for managing resources in an environment in which other processors and other devices can access external memory. V

  Implementations that conform to the VEA also conform to the UISA but may not necessarily adhere to the OEA.
- Operating environment architecture (OEA)—The OEA defines supervisor-level resources typically required by an operating system. It defines the memory management model, supervisor-level registers, and the interrupt model. O

  Implementations that conform to the OEA also conform to the UISA and VEA.

Note that some resources are defined more generally at one level in the architecture and more specifically at another. For example, exception conditions that cause a floating-point interrupt are defined by the UISA, but the interrupt mechanism itself is defined by the OEA.

Because it is important to distinguish between the levels of the architecture to ensure compatibility across multiple platforms, those distinctions are shown clearly throughout this book. The level of the architecture to which text refers is indicated in the outer margin, using the conventions shown in the section "Conventions" at the end of this preface.

For ease in reference, topics in the user's manuals are presented in the same order in this book. Topics build upon one another, beginning with a description and complete summary of the MPC750 programming model (registers and instructions) and progressing to more specific, architecture-based topics regarding the cache, interrupt, and memory management models. As such, chapters may include information from multiple levels of the architecture. For example, the discussion of the cache model uses information from both the VEA and the OEA.

The PowerPC Architecture: A Specification for a New Family of RISC Processors defines the architecture from the perspective of the three programming environments and remains the defining document for the PowerPC architecture.

Information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. As with any technical documentation, it is the readers' responsibility to be sure they are using the most recent version of the documentation.

Audience

It is assumed that the reader understands operating systems, microprocessor system design, and the basic principles of RISC processing.

Organization

Following is a summary and a brief description of the major sections of this manual:

- Chapter 1, "Overview," is useful for those who want a general understanding of the features and functions of the PowerPC architecture. This chapter describes the flexible nature of the PowerPC architecture definition and provides an overview of how the PowerPC architecture defines the register set, operand conventions, addressing modes, instruction set, cache model, interrupt model, and memory management model.
- Chapter 2, "Register Set," is useful for software engineers who need to understand the PowerPC programming model for the three programming environments and the functionality of each register.
- Chapter 3, "Operand Conventions," describes conventions for storing data in memory, including information regarding alignment, single- and double-precision floating-point conventions, and big- and little-endian byte ordering.
- Chapter 4, "Addressing Modes and Instruction Set Summary," provides an overview of the addressing modes and a description of the instructions. Instructions are organized by function.
- Chapter 5, "Cache Model and Memory Coherency," provides a discussion of the cache and memory model defined by the VEA and aspects of the cache model that are defined by the OEA.
- Chapter 6, "Interrupts," describes the interrupt model defined in the OEA.
- Chapter 7, "Memory Management," provides descriptions of the address translation and memory protection mechanism as defined by the OEA.
- Chapter 8, "Instruction Set," functions as a handbook for the instruction set. Instructions are sorted by mnemonic. Each instruction description includes the instruction formats and an individualized legend that provides such information as the level or levels of the architecture in which the instruction may be found and the privilege level of the instruction.

- Appendix A, “Instruction Set Listings,” describes each instruction in detail. Instructions are grouped according to mnemonic, opcode, function, and form.
- Appendix B, “Multiple-Precision Shifts,” describes how multiple-precision shift operations can be programmed as defined by the UISA.
- Appendix C, “Floating-Point Models,” gives examples of how the floating-point conversion instructions can be used to perform various conversions as described in the UISA.
- Appendix D, “Synchronization Programming Examples,” gives examples showing how synchronization instructions can be used to emulate various synchronization primitives and how to provide more complex forms of synchronization.
- Appendix E, “Simplified Mnemonics for PowerPC Instructions,” provides a set of simplified mnemonic examples and symbols.
- Appendix F, “Revision History,” describes major changes since the previous revision of this document.
- This manual also includes a glossary and an index.

Suggested Reading

This section lists additional reading that provides background for the information in this manual as well as general information about the architecture.

General Information

The following documentation, available through Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information:

- *Computer Architecture: A Quantitative Approach*, Third Edition, by John L. Hennessy and David A. Patterson
- *Computer Organization and Design: The Hardware/Software Interface*, Third Edition, David A. Patterson and John L. Hennessy

Related Documentation

Freescale documentation is available from the sources listed on the back cover of this manual; the document order numbers are included in parentheses for ease in ordering:

- *EREF: A Programmer's Reference Manual for Freescale Book E Processors*—This book provides a higher-level view of the programming model as it is defined by Book E and the Freescale Book E implementation standards.
- Reference manuals (formerly called user's manuals)—These books provide details about individual implementations.
- Addenda/errata to reference or user's manuals—Because some processors have follow-on parts an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding reference or user's manuals.
- Hardware specifications—Hardware specifications provide specific data regarding bus timing, signal behavior, and AC, DC, and thermal characteristics, as well as other design considerations.
- Technical summaries—Each device has a technical summary that provides an overview of its features. This document is roughly equivalent to the overview (Chapter 1) of an implementation's reference or user's manual.
- Application notes—These short documents address specific design issues useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to www.freescale.com.

Conventions

This document uses the following notational conventions:

cleared/set	When a bit takes the value zero, it is said to be cleared; when it takes a value of one, it is said to be set.
mnemonics	Instruction mnemonics are shown in lowercase bold.
italics	Italics indicate variable command parameters, for example, **bcctr***x*. Book titles in text are set in italics Internal signals are set in italics, for example, $\overline{qual\ BG}$
0x0	Prefix to denote hexadecimal number
0b0	Prefix to denote binary number
rA, **r**B	Instruction syntax used to identify a source GPR
rD	Instruction syntax used to identify a destination GPR
frA, **fr**B, **fr**C	Instruction syntax used to identify a source FPR
frD	Instruction syntax used to identify a destination FPR
REG[FIELD]	Abbreviations for registers are shown in uppercase text. Specific bits, fields, or ranges appear in brackets. For example, MSR[LE] refers to the little-endian mode enable bit in the machine state register.

x	In some contexts, such as signal encodings, an unitalicized x indicates a don’t care.
x	An italicized *x* indicates an alphanumeric variable.
n	An italicized *n* indicates an numeric variable.
¬	NOT logical operator
&	AND logical operator
|	OR logical operator
||	Concatenation operator; for example, 010 || 111 is the same as 010111.
—	Indicates reserved bits or bit fields in a register. Although these bits can be written to as ones or zeros, they are always read as zeros.
	The following symbols are used both for information that can be found in the architecture specifications as well as for explanatory information related to that programming environment.
U	This symbol identifies text that is relevant with respect to the user instruction set architecture (UISA).
V	This symbol identifies text that is relevant with respect to the virtual environment architecture (VEA).
O	This symbol identifies text that is relevant with respect to the operating environment architecture (OEA).

Additional conventions used with instruction encodings are described in Table 8-2. Conventions used for pseudocode examples are described in Table 8-3.

Acronyms and Abbreviations

Table i contains acronyms and abbreviations that are used in this document. Note that the meanings for some acronyms (such as SDR1 and XER) are historical, and the words for which an acronym stands may not be intuitively obvious.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
ALU	Arithmetic logic unit
BAT	Block address translation
BIST	Built-in self test
BPU	Branch processing unit
BUID	Bus unit ID
CR	Condition register
CTR	Count register
DABR	Data address breakpoint register
DAR	Data address register

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
DBAT	Data BAT
DEC	Decrementer register
DSISR	Register used for determining the source of a DSI interrupt
DTLB	Data translation lookaside buffer
EA	Effective address
EAR	External access register
ECC	Error checking and correction
FPECR	Floating-point exception cause register
FPR	Floating-point register
FPSCR	Floating-point status and control register
FPU	Floating-point unit
GPR	General-purpose register
IBAT	Instruction BAT
IEEE	Institute of Electrical and Electronics Engineers
ITLB	Instruction translation lookaside buffer
IU	Integer unit
L2	Secondary cache
LIFO	Last-in-first-out
LR	Link register
LRU	Least recently used
LSB	Least-significant byte
lsb	Least-significant bit
MESI	Modified/exclusive/shared/invalid—cache coherency protocol
MMU	Memory management unit
MSB	Most-significant byte
msb	Most-significant bit
MSR	Machine state register
NaN	Not a number
NIA	Next instruction address
No-op	No operation
OEA	Operating environment architecture
PIR	Processor identification register
PTE	Page table entry

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
PTEG	Page table entry group
PVR	Processor version register
RISC	Reduced instruction set computing
RTL	Register transfer language
SDR1	Register that specifies the page table base address for virtual-to-physical address translation
SIMM	Signed immediate value
SPR	Special-purpose register
SPRG*n*	Registers available for general purposes
SR	Segment register
SRR0	Machine status save/restore register 0
SRR1	Machine status save/restore register 1
TB	Time base register
TLB	Translation lookaside buffer
UIMM	Unsigned immediate value
UISA	User instruction set architecture
VA	Virtual address
VEA	Virtual environment architecture
XATC	Extended address transfer code
XER	Register used primarily for indicating conditions such as carries and overflows for integer operations

Terminology Conventions

Table ii lists certain terms used in this manual that differ from the architecture terminology conventions.

Table ii. Terminology Conventions

The Architecture Specification	This Manual
Extended mnemonics	Simplified mnemonics
Privileged mode (or privileged state)	Supervisor-level privilege
Problem mode (or problem state)	User-level privilege
Real address	Physical address
Relocation	Translation
Out-of-order memory accesses	Speculative memory accesses
Storage (locations)	Storage or memory
Storage (the act of)	Access

Table iii describes instruction field notation conventions used in this manual.

Table iii. Instruction Field Conventions

The Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D
BF, BFA	**crf**D, **crf**S
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S
SI	SIMM
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Chapter 1
Overview

The architecture provides a software model that ensures software compatibility among implementations. The term ‘implementation’ is used to refer to a hardware device (typically a microprocessor) that complies with the architecture specifications.

The PowerPC architecture is a 64-bit architecture with a 32-bit subset. This manual describes the architecture from a 32-bit perspective. Although some 64-bit resources are discussed, this manual does not completely describe details of the 64-bit–only features of the architecture, in particular with respect to the memory management model, registers, and instruction set.

The architecture defines the following major components:

- Instruction set—The instruction set specifies the families of instructions (such as load/store, integer arithmetic, and floating-point arithmetic instructions), the specific instructions, and the forms used for encoding the instructions. The instruction set definition also specifies the addressing modes used for accessing memory.
- Programming model—The programming model defines the register set and the memory conventions, including details regarding the bit and byte ordering, and the conventions for how data (such as integer and floating-point values) are stored.
- Memory model—The memory model defines the size of the address space and of the subdivisions (pages and blocks) of that address space. It also defines the ability to configure pages and blocks of memory with respect to caching, byte ordering (big- or little-endian), coherency, and various types of memory protection.
- Interrupt model—The interrupt model defines the common set of interrupts and the exception conditions that can generate those interrupts. The interrupt model specifies characteristics of the interrupts, such as whether they are precise or imprecise, synchronous or asynchronous, and maskable or nonmaskable. The interrupt model defines the interrupt vectors and a set of registers used when interrupts are taken. The interrupt model also provides memory space for implementation-specific interrupts.
- Memory management model—The memory management model defines how memory is partitioned, configured, and protected. The memory management model also specifies how memory translation is performed, the real, virtual, and physical address spaces, special memory control instructions, and other characteristics. (Physical address is referred to as real address in the architecture specification.)
- Time-keeping model—The time-keeping model defines facilities that permit the time of day to be determined and the resources and mechanisms required for supporting time-related interrupts.

These aspects of the architecture are defined at different levels of the architecture, and this chapter provides an overview of those levels—the user instruction set architecture (UISA), the virtual environment architecture (VEA), and the operating environment architecture (OEA).

For updates to this document, refer to http://www.freescale.com.

1.1 PowerPC Architecture Overview

The PowerPC architecture, developed jointly by Motorola, IBM, and Apple Computer, is based on the POWER architecture implemented by RS/6000™ family of computers. The PowerPC architecture takes advantage of recent technological advances in such areas as process technology, compiler design, and reduced instruction set computing (RISC) microprocessor design to provide software compatibility across a diverse family of implementations, primarily single-chip microprocessors, intended for a wide range of systems, including battery-powered personal computers; embedded controllers; high-end scientific and graphics workstations; and multiprocessing, microprocessor-based mainframes.

To provide a single architecture for such a broad assortment of processor environments, the PowerPC architecture is both flexible and scalable.

Designers can choose whether to implement architecturally-defined features in hardware or in software. For example, a processor designed for a high-end workstation has greater need for the performance gained from implementing floating-point normalization and denormalization in hardware than a battery-powered, general-purpose computer might.

The architecture is scalable to take advantage of continuing technological advances—for example, the continued miniaturization of transistors makes it more feasible to implement more execution units and a richer set of optimizing features without being constrained by the architecture.

The PowerPC architecture defines the following features:

- Separate 32-entry register files for integer and floating-point instructions. The general-purpose registers (GPRs) hold source data for integer arithmetic instructions, and the floating-point registers (FPRs) hold source and target data for floating-point arithmetic instructions.
- Instructions for loading and storing data between the memory system and either the FPRs or GPRs
- Uniform-length instructions to allow simplified instruction pipelining and parallel processing instruction dispatch mechanisms
- Nondestructive use of registers for arithmetic instructions in which the second, third, and sometimes the fourth operand, typically specify source registers for calculations whose results are typically stored in the target register specified by the first operand.
- A precise interrupt model (with the option of treating floating-point exceptions imprecisely)
- Floating-point support that includes IEEE-754 floating-point operations
- A flexible architecture definition that allows certain features to be performed in either hardware or with assistance from implementation-specific software depending on the needs of the processor design
- The ability to perform both single- and double-precision floating-point operations

- User-level instructions for explicitly storing, flushing, and invalidating data in the on-chip caches. The architecture also defines special instructions (cache block touch instructions) for speculatively loading data before it is needed, reducing the effect of memory latency.
- Definition of a memory model that allows weakly-ordered memory accesses. This allows bus operations to be reordered dynamically, which improves overall performance and in particular reduces the effect of memory latency on instruction throughput.
- Support for separate instruction and data caches (Harvard architecture) and for unified caches
- Support for both big- and little-endian addressing modes

This chapter provides an overview of the major characteristics of the architecture in the order in which they are addressed in this book:

- Register set and programming model
- Instruction set and addressing modes
- Cache implementations
- Interrupt model
- Memory management

1.1.1 The Levels of the PowerPC Architecture

The architecture is defined in three levels that correspond to three programming environments, roughly described from the most general, user-level instruction set environment, to the more specific, operating environment.

This layering of the architecture provides flexibility, allowing degrees of software compatibility across a wide range of implementations. For example, an implementation such as an embedded controller may support the user instruction set, whereas it may be impractical for it to adhere to the memory management, interrupt, and cache models.

The three levels of the architecture are defined as follows:

- User instruction set architecture (UISA—The UISA defines the level of the architecture to which user-level (referred to as problem state in the architecture specification) software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and interrupt model as seen by user programs, and the memory and programming models. The icon shown in the margin identifies text that is relevant with respect to the UISA. U
- Virtual environment architecture (VEA)—The VEA defines additional user-level functionality that falls outside typical user-level software requirements. The VEA describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time base facility from a user-level perspective. The icon shown in the margin identifies text that is relevant with respect to the VEA. V

  Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA.
- Operating environment architecture (OEA)—The OEA defines supervisor-level (referred to as privileged state in the architecture specification) resources typically required by an operating O

system. The OEA defines the memory management model, supervisor-level registers, synchronization requirements, and the interrupt model. The OEA also defines the time base feature from a supervisor-level perspective. The icon shown in the margin identifies text that is relevant with respect to the OEA.

Implementations that adhere to the VEA level are guaranteed to adhere to the UISA level; likewise, implementations that conform to the OEA level are also guaranteed to conform to the UISA and the VEA levels.

All PowerPC devices adhere to the UISA, offering compatibility among all PowerPC application programs. However, there may be different versions of the VEA and OEA than those described here. For example, some devices, such as embedded controllers, may not require some of the features as defined by this VEA and OEA, and may implement a simpler or modified version of those features.

The distinctions between the levels of the PowerPC architecture are maintained clearly throughout this document, using the conventions described in the “Conventions” section of the Preface.

1.1.2 Latitude within the Levels of the Architecture

The architecture defines those parameters necessary to ensure compatibility among processors, but also allows a wide range of options for individual implementations. These are as follows:

- The architecture defines some facilities (such as registers, bits within registers, instructions, and interrupts) as optional.
- The architecture allows implementations to define additional privileged special-purpose registers (SPRs), interrupts, and instructions for special system requirements (such as power management in processors designed for very low-power operation).
- There are many other parameters that the architecture allows implementations to define. For example, the architecture may define conditions for which an interrupt may be taken, such as alignment conditions. A particular implementation may choose to solve the alignment problem without taking the interrupt.
- Processors may implement any architectural facility or instruction with assistance from software (that is, they may trap and emulate) as long as the results (aside from performance) are identical to that specified by the architecture.
- Some parameters are defined at one level of the architecture and defined more specifically at another. For example, the UISA defines conditions that may cause an alignment interrupt, and the OEA specifies the interrupt itself.

1.1.3 Features Not Defined by the PowerPC Architecture

Because flexibility is an important design goal of the PowerPC architecture, there are many aspects of the processor design, typically relating to the hardware implementation, that the PowerPC architecture does not define, such as the following:

- System bus interface signals—Although numerous implementations may have similar interfaces, the PowerPC architecture does not define individual signals or the bus protocol. For example, the OEA allows each implementation to determine the signal or signals that trigger the machine check interrupt.

- Cache design—The PowerPC architecture does not define the size, structure, the replacement algorithm, or the mechanism used for maintaining cache coherency. The PowerPC architecture supports, but does not require, the use of separate instruction and data caches. Likewise, the PowerPC architecture does not specify the method by which cache coherency is ensured.
- The number and the nature of execution units—The PowerPC architecture is a RISC architecture, and as such has been designed to facilitate the design of processors that use pipelining and parallel execution units to maximize instruction throughput. However, the PowerPC architecture does not define the internal hardware details of implementations. For example, one processor may execute load and store operations in the integer unit, while another may execute these instructions in a dedicated load/store unit.
- Other internal microarchitecture issues—The architecture does not prescribe which execution unit is responsible for executing a particular instruction; it also does not define details regarding the instruction fetching mechanism, how instructions are decoded and dispatched, and how results are written back. Dispatch and write-back may occur in order or out of order. Also while the architecture specifies certain registers, such as the GPRs and FPRs, implementations can implement register renaming or other schemes to reduce the impact of data dependencies and register contention.

1.2 The Architectural Models

This section provides overviews of aspects defined by the architecture, following the same order as the rest of this book. The topics include the following:

- Registers and programming model
- Operand conventions
- Instruction set and addressing modes
- Cache model
- Interrupt model
- Memory management model

1.2.1 Registers and Programming Model

The architecture defines register-to-register operations for computational instructions. Source operands for these instructions are accessed from the architected registers or are provided as immediate values embedded in the instruction. The three-register instruction format allows specification of a target register distinct from two source operand registers. This scheme allows efficient code scheduling in a highly parallel processor. Load and store instructions are the only instructions that transfer data between registers and memory. The PowerPC registers are shown in Figure 1-1.

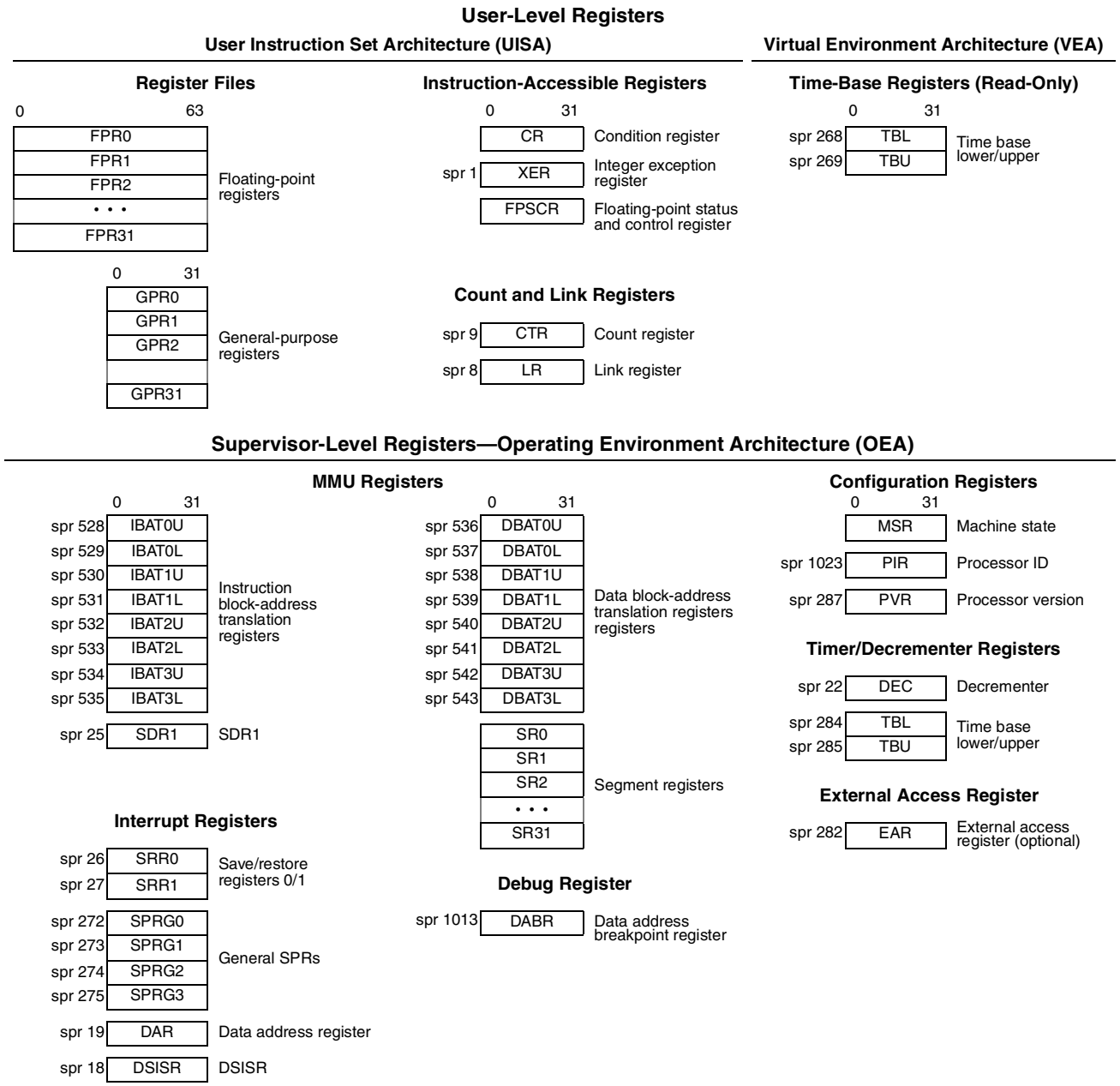


Figure 1-1. Programming Model—PowerPC Registers

The programming model incorporates 32 GPRs, 32 FPRs, special-purpose registers (SPRs), and several miscellaneous registers. Each implementation typically has registers that are not defined by the architecture.

PowerPC processors have two levels of privilege:

- Supervisor mode—used exclusively by the operating system. Resources defined by the OEA can be accessed only supervisor-level software.
- User mode—used by the application software and operating system software. Only resources defined by the UISA and VEA can be accessed by user-level software.

These two levels govern the access to registers, as shown in Figure 1-1. The division of privilege allows the operating system to control the application environment (providing virtual memory and protecting operating system and critical machine resources). Instructions that control the state of the processor, the address translation mechanism, and supervisor registers can be executed only when the processor is operating in supervisor mode.

- User instruction set architecture registers—All UISA registers can be accessed by all software with either user or supervisor privileges. These registers include the 32 general-purpose registers (GPRs) and the 32 floating-point registers (FPRs), and other registers used for integer, floating-point, and branch instructions. U
- Virtual environment architecture registers—The VEA defines the user-level portion of the time base facility, which consists of the two 32-bit time base registers. These registers can be read by user-level software, but can be written to only by supervisor-level software. V
- Operating environment architecture registers—SPRs defined by the OEA are used for system-level operations such as memory management, interrupt handling, and time keeping. O

The architecture also provides room in the SPR space for implementation-specific registers, which are not discussed in this manual.

1.2.2 Operand Conventions

Operand conventions are defined in two levels of the architecture—UISA and VEA. These conventions define how data is stored in registers and memory. U V

1.2.2.1 Byte Ordering

The default mapping for processors is big-endian, but the UISA provides the option of operating in either big- or little-endian mode. Big-endian byte ordering is shown in Figure 1-2. U

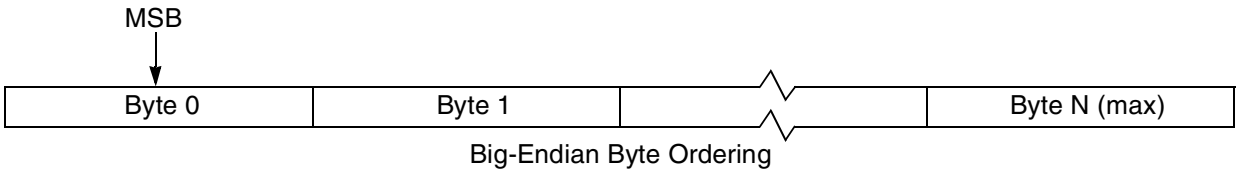


Figure 1-2. Big-Endian Byte and Bit Ordering

The OEA defines two bits in the MSR for specifying byte ordering—LE (little-endian mode) and ILE (interrupt little-endian mode). MSR[LE] specifies whether the processor is configured for big-endian or little-endian mode; MSR[ILE] specifies the mode when an interrupt is taken by being copied into MSR[LE]. A value of 0 specifies big-endian mode and a value of 1 specifies little-endian mode. O

1.2.2.2 Data Organization in Memory and Data Transfers

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte.

Memory operands may be bytes, half words, words, or double words, or, for the load/store string/multiple instructions, a sequence of bytes or words. The address of a multiple-byte memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the natural address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned.

1.2.2.3 Floating-Point Conventions

U The architecture adheres to the IEEE-754 standard for 64- and 32-bit floating-point arithmetic:

- Double-precision arithmetic instructions may have single- or double-precision operands but always produce double-precision results.
- Single-precision arithmetic instructions require all operands to be single-precision values and always produce single-precision results. Single-precision values are stored in double-precision format in the FPRs—these values are rounded such that they can be represented in 32-bit, single-precision format (as they are in memory).

1.2.3 Instruction Set and Addressing Modes

All instructions are encoded as single-word (32-bit) instructions. Instruction formats are consistent among all instruction types, permitting decoding to occur in parallel with operand accesses. This fixed instruction length and consistent format greatly simplifies instruction pipelining.

1.2.3.1 Instruction Set

Although these categories are not defined by the architecture, the instructions can be grouped as follows:

U
- Integer instructions—These computational and logical instructions are defined by the UISA.
  - — Integer arithmetic instructions
  - — Integer compare instructions
  - — Logical instructions
  - — Integer rotate and shift instructions
- Floating-point instructions—These instructions, defined by the UISA, include floating-point computational instructions, as well as instructions that manipulate the floating-point status and control register (FPSCR).
  - — Floating-point arithmetic instructions
  - — Floating-point multiply/add instructions
  - — Floating-point compare instructions

  - Floating-point status and control instructions
  - Floating-point move instructions
  - Optional floating-point instructions
- Load/store instructions—These instructions, defined by the UISA, include integer and floating-point load and store instructions.
  - Integer load and store instructions
  - Integer load and store with byte-reverse instructions
  - Integer load and store multiple instructions
  - Integer load and store string instructions
  - Floating-point load and store instructions
- The UISA also provides a set of load/store with reservation instructions (**lwarx** and **stwcx**.) that can be used as primitives for constructing atomic memory operations. These are grouped under synchronization instructions.
- Synchronization instructions—The UISA and VEA define instructions for memory synchronizing, especially useful for multiprocessing:
  - Load and store with reservation instructions—These UISA-defined instructions provide primitives for synchronization operations such as test and set, compare and swap, and compare memory.
  - The Synchronize instruction (**sync**)—This UISA-defined instruction is useful for synchronizing load and store operations on a memory bus that is shared by multiple devices.
  - Enforce In-Order Execution of I/O (**eieio**)—The **eieio** instruction provides an ordering function for the effects of load and store operations executed by a processor.
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow.
  - The UISA defines numerous instructions that control the program flow, including branch, trap, and system call instructions as well as instructions that read, write, or manipulate bits in the condition register. U
  - The OEA defines two flow control instructions that provide system linkage. These instructions are used for entering and returning from supervisor level. O
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing caches and translation lookaside buffers (TLBs) (and segment registers in 32-bit implementations). These instructions include move to/from special-purpose register instructions (**mtspr** and **mfspr**).
- Memory/cache control instructions—These instructions provide control of caches, TLBs, and segment registers.
  - The VEA defines several cache control instructions. V
  - The OEA defines one cache control instruction and several memory control instructions. O
- External control instructions—The VEA defines two optional instructions for use with special input/output devices. V

Note that this grouping of the instructions does not indicate which execution unit executes a particular instruction or group of instructions. This is not defined by the architecture.

1.2.3.2 Calculating Effective Addresses

U The effective address (EA), also called the logical address, is the address computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. Unless address translation is disabled, this address is converted by the MMU to the appropriate physical address. (Note that the architecture specification uses only the term effective address and not logical address.)

The architecture supports the following simple addressing modes for memory access instructions:

- EA = (**r**A|0) (register indirect)
- EA = (**r**A|0) + offset (including offset = 0) (register indirect with immediate index)
- EA = (**r**A|0) + **r**B (register indirect with index)

These simple addressing modes allow efficient address generation for memory accesses.

1.2.4 Cache Model

V The VEA and OEA define aspects of cache implementations for processors. The architecture does not define hardware aspects of cache implementations. For example, some processors may have separate O instruction and data caches (Harvard architecture), while others have a unified cache.

The architecture allows implementations to control the following memory access modes on a page or block basis:

- Write-back/write-through mode
- Caching-inhibited mode
- Memory coherency
- Guarded/not guarded against speculative accesses

Coherency is maintained on a cache block basis, and cache control instructions perform operations on a cache block basis. The size of the cache block is implementation-dependent. The term cache block should not be confused with the notion of a block in memory, which is described in Section 1.2.6, “Memory Management Model (MMU).”

The VEA defines several instructions for cache management. These can be used by user-level software to perform such operations as touch operations (which cause the cache block to be speculatively loaded), and O operations to store, flush, or clear the contents of a cache block. The OEA portion of the architecture defines one cache management instruction—the Data Cache Block Invalidate (**dcbi**) instruction.

1.2.5 Interrupt Model

The interrupt mechanism, defined by the OEA, allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When interrupts occur, information about the state of the processor is saved to various registers and the processor begins execution at an address (interrupt vector) predetermined for each type of interrupt. Interrupt handler

routines begin execution in supervisor mode. The interrupt model is described in detail in Chapter 6, “Interrupts.” Note also that some aspects regarding interrupt conditions are defined at other levels of the architecture. For example, floating-point exception conditions are defined by the UISA, whereas the interrupt mechanism is defined by the OEA.

The architecture requires that interrupts be handled in program order (excluding the optional floating-point imprecise modes and the reset and machine check interrupt); therefore, although a particular implementation may recognize interrupt conditions out of order, they are handled strictly in order. When an instruction-caused interrupt is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet begun to execute, are required to complete before the interrupt is taken. Any interrupts caused by those instructions must be handled first. Likewise, interrupts that are asynchronous and precise are recognized when they occur, but are not handled until all instructions currently executing successfully complete processing and report their results.

The OEA supports four types of interrupts:

- Synchronous, precise
- Synchronous, imprecise
- Asynchronous, maskable
- Asynchronous, nonmaskable

1.2.6 Memory Management Model (MMU)

The MMU specifications are largely provided by the OEA. The primary functions of the MMU are to translate logical (effective) addresses to physical addresses for memory accesses and I/O accesses (most I/O accesses are assumed to be memory-mapped), and to provide access protection on a block or page basis. Note that many aspects of memory management are implementation-dependent. The description in Chapter 7, “Memory Management,” describes the conceptual model of a MMU; however, processors may differ in the specific hardware used to implement the MMU model.

Processors require address translation for two types of transactions—instruction accesses and data accesses to memory (typically generated by load and store instructions).

The memory management specification includes models for 32-bit implementations. The MMU of a 32-bit processor provides 2^{32} bytes of logical address space accessible to supervisor and user programs with a 4-Kbyte page size and 256-Mbyte segment size.

In 32-bit implementations, the entire 4-Gbyte memory space is defined by sixteen 256-Mbyte segments. Segments are configured through the 16 segment registers.

The block address translation (BAT) mechanism maps large blocks of memory. Block sizes range from 128 Kbytes to 256 Mbytes and are software-selectable. In addition, the MMU of 32-bit processors uses an interim virtual address (52 bits) and hashed page tables in the generation of 32-bit physical addresses.

Two types of processor-generated accesses require address translation: instruction accesses and data accesses to memory generated by load and store instructions. The address translation mechanism is defined in terms of segment tables (or segment registers in 32-bit implementations) and page tables used to locate the logical-to-physical address mapping for instruction and data accesses. The segment

information translates the logical address to an interim virtual address, and the page table information translates the virtual address to a physical address.

Translation lookaside buffers (TLBs) are commonly implemented to keep recently-used page table entries on-chip. Although their exact characteristics are not specified by the architecture, the general concepts that are pertinent to the system software are described.

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor SPRs; refer to Chapter 7, “Memory Management,” for more information.

Chapter 2
Register Set

This chapter describes the register organization defined by the three levels of the architecture—user instruction set architecture (UISA, virtual environment architecture (VEA), and operating environment architecture (OEA). The architecture defines register-to-register operations for all computational instructions. Source data for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the opcode. The three-register instruction format allows specification of a target register distinct from the two source registers, preserving the original data for use by other instructions and reducing the number of instructions for some operations. Data is transferred between memory and registers with explicit load and store instructions only.

Note that the handling of reserved bits in any register is implementation-dependent. Software is permitted to write any value to a reserved bit in a register. However, a subsequent reading of the reserved bit returns 0 if the value last written to the bit was 0 and returns an undefined value (may be 0 or 1) otherwise. This means that even if the last value written to a reserved bit was 1, reading that bit may return 0.

2.1 UISA Register Set

The UISA registers, shown in Figure 2-1, can be accessed by either user- or supervisor-level instructions (the architecture specification refers to user-level and supervisor-level as problem state and privileged state respectively). The general-purpose registers (GPRs) and floating-point registers (FPRs) are accessed as instruction operands. Access to registers can be explicit (that is, through the use of specific instructions for that purpose such as Move to Special-Purpose Register (**mtspr**) and Move from Special-Purpose Register (**mfspr**) instructions) or implicit as part of the execution of an instruction. Some registers are accessed both explicitly and implicitly.

The number to the right of the register names indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR 1).

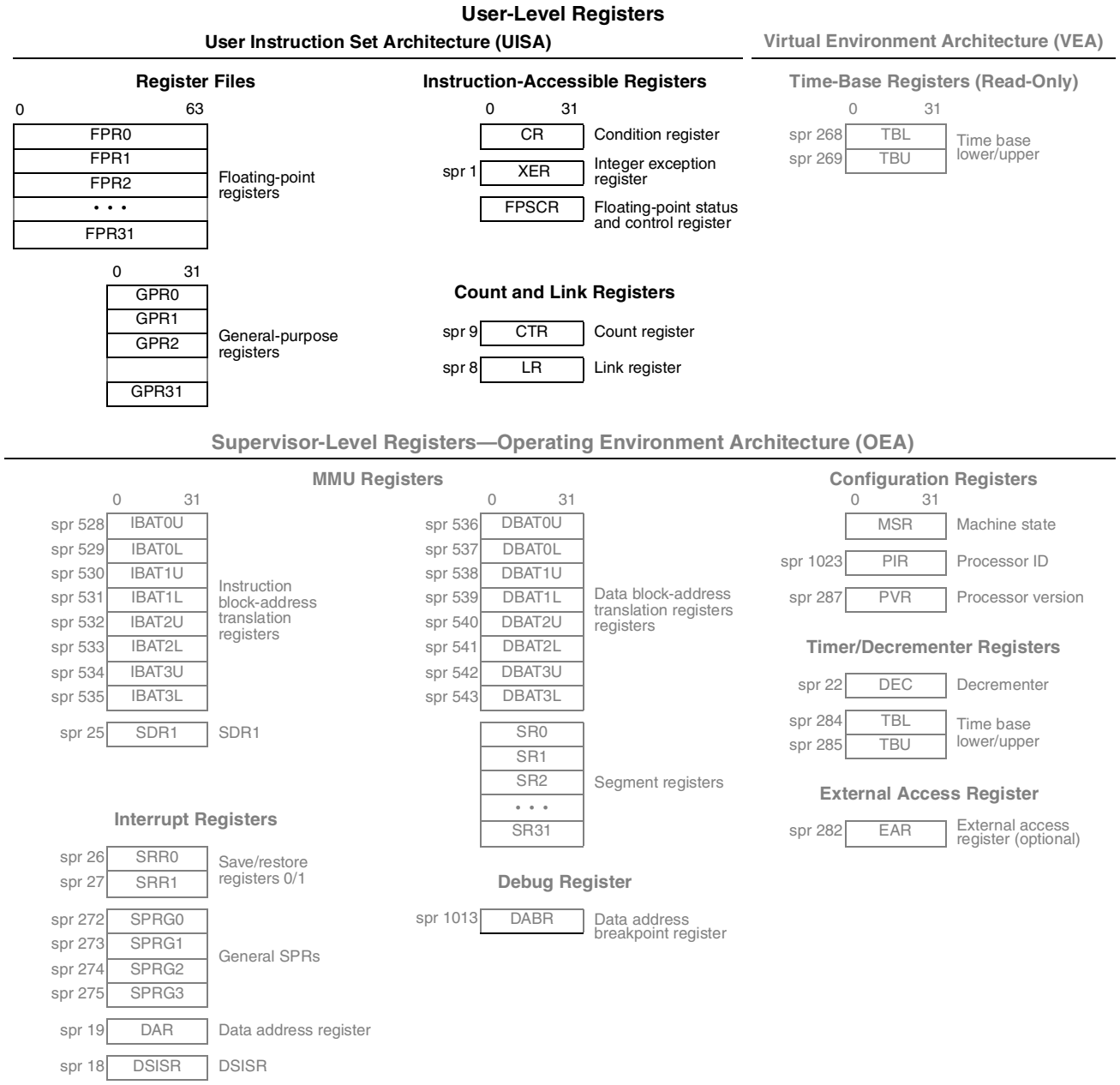


Figure 2-1. UISA Programming Model—User-Level Registers

The user-level registers can be accessed by all software with either user or supervisor privileges. The user-level register set includes the following:

- General-purpose registers (GPRs). The general-purpose register file consists of 32 GPRs designated as GPR0–GPR31. The GPRs serve as data source or destination registers for all integer instructions and provide data for generating addresses. See Section 2.1.1, “General-Purpose Registers (GPRs).”

- Floating-point registers (FPRs). The floating-point register file consists of 32 FPRs designated as FPR0–FPR31; FPRs serve as the data source or destination for all floating-point instructions. The floating-point model includes data objects of either single- or double-precision floating-point format, but the FPRs only contain data in double-precision format. See Section 2.1.2, “Floating-Point Registers (FPRs).”
- Condition register (CR). The 32-bit CR has eight 4-bit fields, CR0–CR7, that reflect the results of certain arithmetic operations and provides a mechanism for testing and branching. See Section 2.1.3, “Condition Register (CR).”
- Floating-point status and control register (FPSCR). The FPSCR contains all signal, summary and enable bits for floating-point exceptions and rounding control bits for compliance with the IEEE 754 standard. See Section 2.1.4, “Floating-Point Status and Control Register (FPSCR).”
- XER register (XER). The XER indicates overflows and carry conditions for integer operations and the number of bytes to be transferred by the load/store string indexed instructions. See Section 2.1.5, “XER Register (XER).”
- Link register (LR). The LR provides the branch target address for the Branch Conditional to Link Register (**bclr***x*) instructions, and can optionally be used to hold the effective address of the instruction that follows a branch with link update instruction in the instruction stream, typically used for loading the return pointer for a subroutine. See Section 2.1.6, “Link Register (LR).”
- Count register (CTR). The CTR holds a loop count that can be decremented during execution of appropriately coded branch instructions. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instructions. See Section 2.1.7, “Count Register (CTR).”

2.1.1 General-Purpose Registers (GPRs)

Integer data is manipulated in the processor’s 32 GPRs shown in Figure 2-2. The GPRs are accessed as source and destination registers in the instruction syntax.

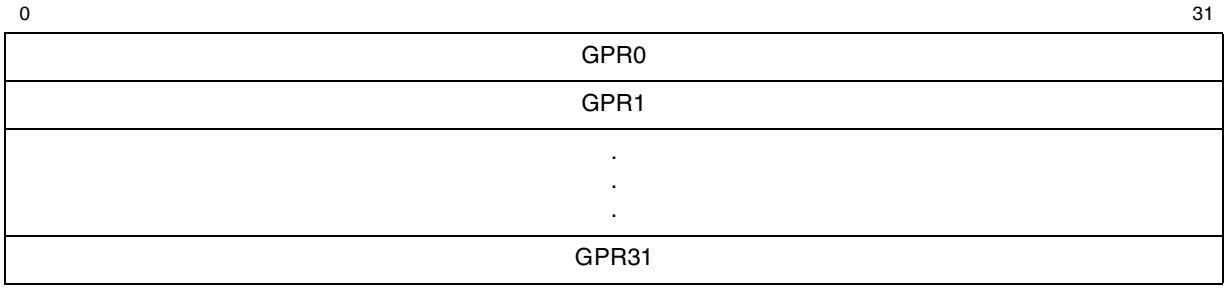


Figure 2-2. General-Purpose Registers (GPRs)

2.1.2 Floating-Point Registers (FPRs)

The architecture provides thirty-two 64-bit FPRs as shown in Figure 2-3. These registers are accessed as source and destination registers for floating-point instructions. Each FPR supports the double-precision floating-point format. Every instruction that interprets the contents of an FPR as a floating-point value uses the double-precision floating-point format for this interpretation.

All floating-point arithmetic instructions operate on data located in FPRs and, with the exception of compare instructions, place the result into an FPR. Information about the status of floating-point operations is placed into the FPSCR and in some cases, into the CR after the completion of instruction execution. For information on how the CR is affected for floating-point operations, see Section 2.1.3, “Condition Register (CR).”

Load and store double-word instructions transfer 64 bits of data between memory and the FPRs with no conversion. Load single instructions are provided to read a single-precision floating-point value from memory, convert it to double-precision floating-point format, and place it in the target floating-point register. Store single-precision instructions are provided to read a double-precision floating-point value from a floating-point register, convert it to single-precision floating-point format, and place it in the target memory location.

Single- and double-precision arithmetic instructions accept values from the FPRs in double-precision format. For single-precision arithmetic and store instructions, all input values must be representable in single-precision format; otherwise, the result placed into the target FPR (or the memory location) and the setting of status bits in the FPSCR and in the condition register (if the instruction’s record bit, Rc, is set) are undefined.

The floating-point arithmetic instructions produce intermediate results that may be regarded as infinitely precise and with unbounded exponent range. This intermediate result is normalized or denormalized if required, and then rounded to the destination format. The final result is then placed into the target FPR in the double-precision format or in fixed-point format, depending on the instruction. Refer to Section 3.3, “Floating-Point Execution Models—UISA.”

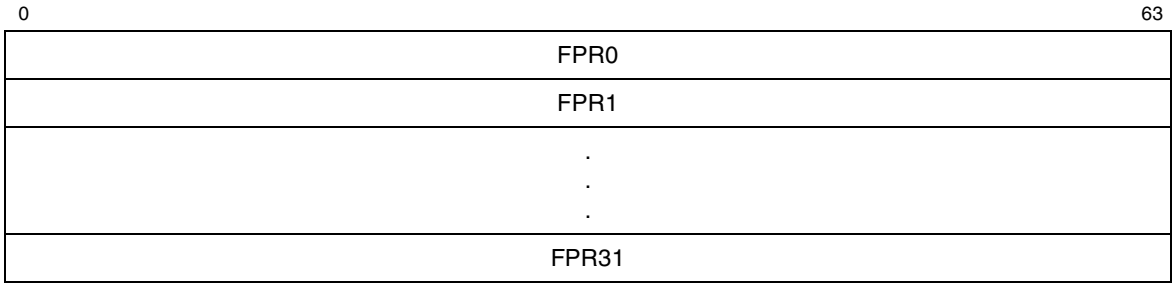


Figure 2-3. Floating-Point Registers (FPRs)

2.1.3 Condition Register (CR)

The 32-bit condition register (CR) reflects the result of certain operations and provides a mechanism for testing and branching. CR bits are grouped into eight 4-bit fields, CR0–CR7, as shown in Figure 2-4.

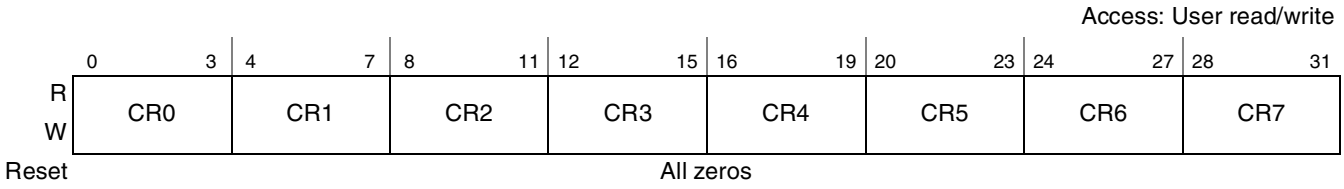


Figure 2-4. Condition Register (CR)

The CR fields can be set by using the following instructions:

- Specified CR fields are set from a GPR by using **mtcrf**.
- The contents of one CR field are copied into another CR field by using **mcrf**. All other condition register fields remain unchanged.
- The contents of XER[0–3] is moved to another CR field by using **mcrxr**.
- A specified FPSCR field is copied to a specified field of the CR by using **mcrfs**.
- CR logical instructions perform logical operations on specified CR bits.
- CR0 can be the implicit result of an integer instruction.
- CR1 can be the implicit result of a floating-point instruction.
- A specified CR field can indicate the result of either an integer or floating-point compare instruction.

Note that branch instructions are provided to test individual CR bits.

2.1.3.1 Condition Register CR0 Field Definition

For all integer instructions, when the CR is set to reflect the result of the operation (that is, when Rc = 1), and for **addic.**, **andi.**, and **andis.**, the first three bits of CR0 are set by an algebraic comparison of the result to zero; the fourth bit of CR0 is copied from XER[SO]. For integer instructions, CR bits 0–3 are set to reflect the result as a signed quantity.

The CR bits are interpreted as shown in Table 2-1. If any portion of the result is undefined, the value placed into the first three bits of CR0 is undefined.

Table 2-1. Bit Settings for CR0 Field of CR

Bits	Description
0	Negative (LT)—This bit is set when the result is negative.
1	Positive (GT)—This bit is set when the result is positive (and not zero).
2	Zero (EQ)—This bit is set when the result is zero.
3	Summary overflow (SO)—This is a copy of the final state of XER[SO] at the completion of the instruction.

Note that CR0 may not reflect the true (that is, infinitely precise) result if overflow occurs.

2.1.3.2 Condition Register CR1 Field Definition

In all floating-point instructions when the CR is set to reflect the result of the operation (that is, when the instruction's record bit, Rc, is set), CR1 (bits 4–7 of the CR) is copied from FPSCR[0–3] and indicates the floating-point exception status. See Section 2.1.4, "Floating-Point Status and Control Register (FPSCR)." Table 2-2 shows CR1 bit settings.

Table 2-2. Bit Settings for CR1 Field of CR

Bits	Description
4	Floating-point exception (FX). Copy of the final state of FPSCR[FX] at the completion of the instruction.
5	Floating-point enabled exception (FEX). Copy of the final state of FPSCR[FEX] at the completion of the instruction.
6	Floating-point invalid exception (VX). Copy of the final state of FPSCR[VX] at the completion of the instruction.
7	Floating-point overflow exception (OX). Copy of the final state of FPSCR[OX] at the completion of the instruction.

2.1.3.3 Condition Register CR*n* Field—Compare Instruction

For a compare instruction, when a specified CR field is set to reflect the result of the comparison, the bits of the specified field are interpreted as shown in Table 2-3.

Table 2-3. CR*n* Field Bit Settings for Compare Instructions

Bits[1]	Description[2]
0	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A < **fr**B.
1	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A > **fr**B.
2	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
3	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of the final state of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a Not a Number (NaN).

1 Here, the bit indicates the bit number in any one of the 4-bit subfields, CR0–CR7.

2 For a complete description of instruction syntax conventions, refer to Table 8-2.

2.1.4 Floating-Point Status and Control Register (FPSCR)

The FPSCR, shown in Figure 2-5, contains bits that do the following:

- Record exceptions generated by floating-point operations
- Record the type of the result produced by a floating-point operation
- Control the rounding mode used by floating-point operations
- Enable or disable the reporting of exceptions (invoking the interrupt handler)

Bits 0–23 are status bits, which are updated at the completion of the instruction execution. Bits 24–31 are control bits.

Except for the floating-point enabled exception summary (FEX) and floating-point invalid operation exception summary (VX), the exception condition bits, FPSCR[0–12,21–23], are sticky. Once set, sticky bits remain set until they are cleared by an **mcrfs**, **mtfsfi**, **mtfsf**, or **mtfsb0** instruction.

FEX and VX are the logical ORs of other FPSCR bits. Therefore, these two bits are not listed among the FPSCR bits directly affected by the various instructions.

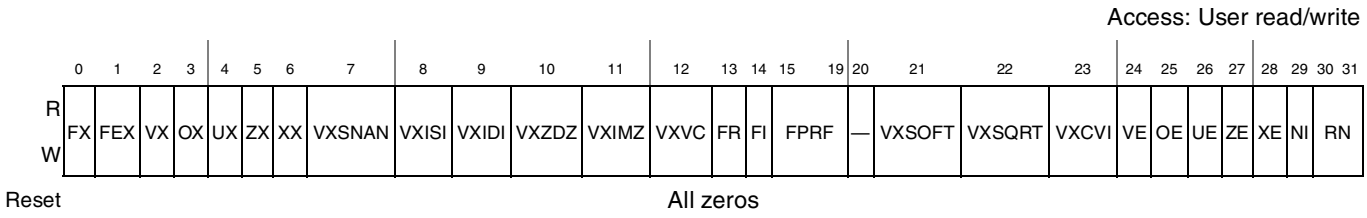


Figure 2-5. Floating-Point Status and Control Register (FPSCR)

FPSCR bits are described in Table 2-4

Table 2-4. FPSCR Bit Settings

Bits	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FX if that instruction causes any FPSCR floating-point exception bit to transition from 0 to 1. The **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** instructions can alter FX explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. Signals the occurrence of any enabled exception conditions. It is the logical OR of all the floating-point exception bits masked by their respective enable bits (FEX = (VX & VE) ^ (OX & OE) ^ (UX & UE) ^ (ZX & ZE) ^ (XX & XE)). The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[FEX] explicitly. This is not a sticky bit.
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exception bits as described in Section 3.3.6.1.1, “Invalid Operation Exception Condition.” The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception. This is a sticky bit. See Section 3.3.6.2, “Overflow, Underflow, and Inexact Exception Conditions.”
4	UX	Floating-point underflow exception. This is a sticky bit. See Section 3.3.6.2.2, “Underflow Exception Condition.”
5	ZX	Floating-point zero divide exception. This is a sticky bit. See Section 3.3.6.1.2, “Zero Divide Exception Condition.”
6	XX	Floating-point inexact exception. This is a sticky bit. See Section 3.3.6.2.3, “Inexact Exception Condition.” XX is the sticky version of FPSCR[FI]. A given instruction sets XX as follows: • If the instruction affects FPSCR[FI], the new value of FPSCR[XX] is obtained by logically ORing the old value of FPSCR[XX] with the new value of FPSCR[FI]. • If the instruction does not affect FPSCR[FI], the value of FPSCR[XX] is unchanged.
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”

Table 2-4. FPSCR Bit Settings (continued)

Bits	Name	Description
8	VXISI	Floating-point invalid operation exception for ∞ – ∞. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
9	VXIDI	Floating-point invalid operation exception for ∞ ÷ ∞. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
10	VXZDZ	Floating-point invalid operation exception for 0 ÷ 0. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
11	VXIMZ	Floating-point invalid operation exception for ∞ * 0. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
13	FR	Floating-point fraction rounded. The last arithmetic, rounding, or conversion instruction incremented the fraction. See Section 3.3.5, “Rounding.” This bit is not sticky.
14	FI	Floating-point fraction inexact. The last arithmetic, rounding, or conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. See Section 3.3.5, “Rounding.” This is not a sticky bit. For more information regarding the relationship between FPSCR[FI] and FPSCR[XX], see the description of the FPSCR[XX] bit.
15–19	FPRF	Floating-point result flags. For arithmetic, rounding, and conversion instructions, FPRF is based on the result placed into the target register, except that if any portion of the result is undefined, the value placed here is undefined. 15 Floating-point result class descriptor (C). Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits to indicate the class of the result as shown in Table 2-5. Bits 16–19 comprise the floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero. 16 Floating-point less than or negative (FL or <) 17 Floating-point greater than or positive (FG or >) 18 Floating-point equal or zero (FE or =) 19 Floating-point unordered or NaN (FU or ?) Note that these are not sticky bits.
20	—	Reserved, should be cleared.
21	VXSOFT	Floating-point invalid operation exception for software request. This is a sticky bit. This bit can be altered only by the **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1** instructions. For more detailed information, refer to Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit. For more detailed information, refer to Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
24	VE	Floating-point invalid operation exception enable. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
25	OE	IEEE floating-point overflow exception enable. See Section 3.3.6.2, “Overflow, Underflow, and Inexact Exception Conditions.”
26	UE	IEEE floating-point underflow exception enable. See Section 3.3.6.2.2, “Underflow Exception Condition.”

Table 2-4. FPSCR Bit Settings (continued)

Bits	Name	Description
27	ZE	IEEE floating-point zero divide exception enable. See Section 3.3.6.1.2, “Zero Divide Exception Condition.”
28	XE	Floating-point inexact exception enable. See Section 3.3.6.2.3, “Inexact Exception Condition.”
29	NI	Floating-point non-IEEE mode. If this bit is set, results need not conform with IEEE standards and the other FPSCR bits may have meanings other than those described here. If the bit is set and if all *i*mplementation-specific requirements are met and if an IEEE-conforming result of a floating-point operation would be a denormalized number, the result produced is zero (retaining the sign of the denormalized number). Any other effects associated with setting this bit are described in the user’s manual for the implementation. Effects of the setting of this bit are implementation-dependent.
30–31	RN	Floating-point rounding control. See Section 3.3.5, “Rounding.” 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

Table 2-5 describes the floating-point result flags, which correspond to FPSCR[15–19].

Table 2-5. Floating-Point Result Flags in FPSCR

Result Flags (Bits 15–19)					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	–Infinity
0	1	0	0	0	–Normalized number
1	1	0	0	0	–Denormalized number
1	0	0	1	0	–Zero
0	0	0	1	0	+Zero
1	0	1	0	0	+Denormalized number
0	0	1	0	0	+Normalized number
0	0	1	0	1	+Infinity

2.1.5 XER Register (XER)

The XER register is a 32-bit, user-level register shown in Figure 2-6.

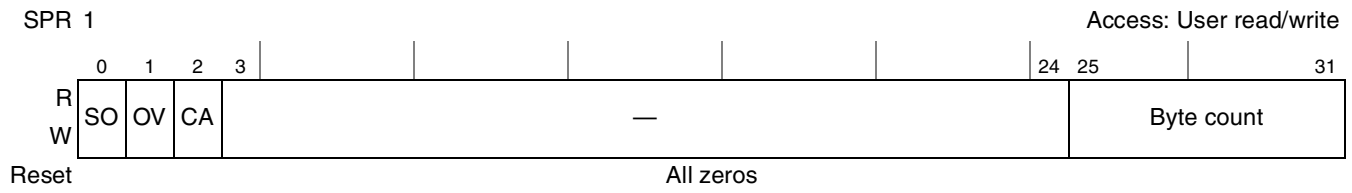


Figure 2-6. XER Register

The XER bit definitions, shown in Table 2-6, are based on the operation of an instruction considered as a whole, not on intermediate results. For example, the result of the Subtract from Carrying (**subfc***x*) instruction is specified as the sum of three values. This instruction sets XER bits based on the entire operation, not on an intermediate sum.

Table 2-6. XER Bit Definitions

Bits	Name	Description
0	SO	Summary overflow. The summary overflow bit (SO) is set whenever an instruction (except **mtspr**) sets the overflow bit (OV). Once set, the SO bit remains set until it is cleared by an **mtspr** instruction (specifying the XER) or an **mcrxr** instruction. It is not altered by compare instructions, nor by other instructions (except **mtspr** to the XER, and **mcrxr**) that cannot overflow. Executing an **mtspr** instruction to the XER, supplying the values zero for SO and one for OV, causes SO to be cleared and OV to be set.
1	OV	Overflow. The overflow bit (OV) is set to indicate that an overflow has occurred during execution of an instruction. Add, subtract from, and negate instructions having OE = 1 set the OV bit if the carry out of the msb is not equal to the carry out of the msb + 1, and clear it otherwise. Multiply low and divide instructions having OE = 1 set the OV bit if the result cannot be represented in 64 bits (**mulld**, **divd**, **divdu**) or in 32 bits (**mullw**, **divw**, **divwu**), and clear it otherwise. The OV bit is not altered by compare instructions that cannot overflow (except **mtspr** to the XER, and **mcrxr**).
2	CA	Carry. Set during execution of the following instructions: • Add carrying, subtract from carrying, add extended, and subtract from extended instructions set CA if there is a carry out of the msb, and clear it otherwise. • Shift right algebraic instructions set CA if any 1 bits have been shifted out of a negative operand, and clear it otherwise. The CA bit is not altered by compare instructions, nor by other instructions that cannot carry (except shift right algebraic, **mtspr** to the XER, and **mcrxr**).
3–24	—	Reserved
25–31	Byte count	This field specifies the number of bytes to be transferred by a Load String Word Indexed (**lswx**) or Store String Word Indexed (**stswx**) instruction.

2.1.6 Link Register (LR)

The link register (LR) supplies the branch target address for the Branch Conditional to Link Register (**bclr***x*) instructions, and in the case of a branch with link update instruction, can be used to hold the logical address of the instruction that follows the branch with link update instruction (for returning from a subroutine). The format of LR is shown in Figure 2-7.

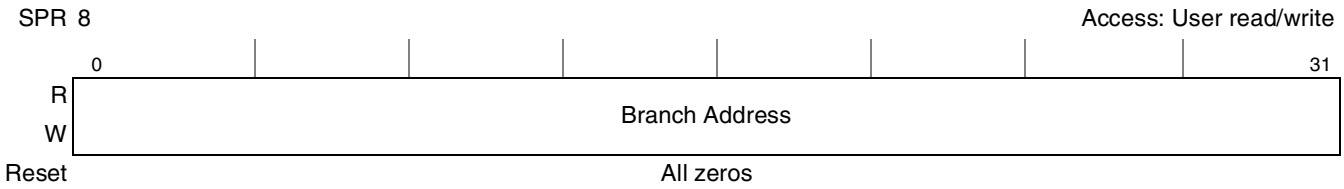


Figure 2-7. Link Register (LR)

Note that although the two least-significant bits can accept any values written to them, they are ignored when the LR is used as an address. Both conditional and unconditional branch instructions include the option of placing the logical address of the instruction following the branch instruction in the LR.

The link register can be also accessed by the **mtspr** and **mfspr** instructions using SPR 8. Prefetching instructions along the target path (loaded by an **mtspr** instruction) is possible provided the LR is loaded sufficiently ahead of the branch instruction (so that any branch prediction hardware can calculate the branch address). Additionally, processors can prefetch along a target path loaded by a branch and link instruction.

Note that some processors may keep a stack of the LR values most recently set by branch with link update instructions. To benefit from these enhancements, use of the LR should be restricted to the manner described in Section 4.2.4.2, “Conditional Branch Control.”

2.1.7 Count Register (CTR)

The count register (CTR) can hold a loop count that can be decremented during execution of branch instructions that contain an appropriately coded BO field. If the value in CTR is 0 before being decremented, it is 0xFFFF_FFFF (2^{32}–1) afterward. The CTR can also provide the branch target address for the Branch Conditional to Count Register (**bcctr***x*) instruction.

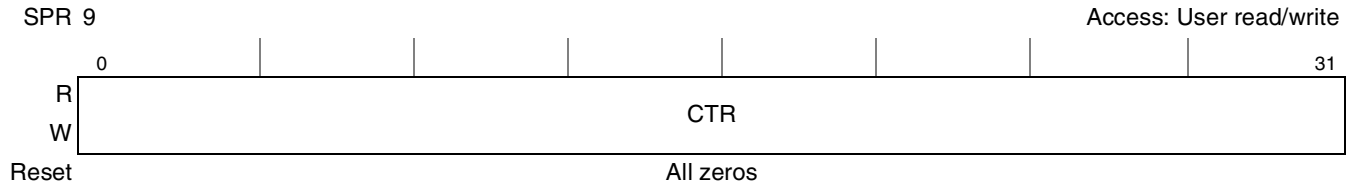


Figure 2-8. Count Register (CTR)

Prefetching instructions along the target path is also possible provided the count register is loaded sufficiently ahead of the branch instruction (so that any branch prediction hardware can calculate the correct value of the loop count).

The count register can also be accessed by the **mtspr** and **mfspr** instructions by specifying SPR 9. In branch conditional instructions, the BO field specifies the conditions under which the branch is taken. The first four bits of the BO field specify how the branch is affected by or affects the CR and the CTR. The encoding for the BO field is shown in Table 2-7.

Table 2-7. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.

Table 2-7. BO Operand Encodings (continued)

BO	Description
1*z*1*zz*	Branch always.

Notes: The *y* bit provides a hint about whether a conditional branch is likely to be taken and is used by some implementations to improve performance. Other implementations may ignore the *y* bit.

The *z* indicates a bit that is ignored. The *z* bits should be cleared (zero), as they may be assigned a meaning in a future version of the UISA.

2.2 VEA Register Set—Time Base

The virtual environment architecture (VEA) defines registers in addition to those defined by the UISA. The VEA register set can be accessed by all software with either user- or supervisor-level privileges. Figure 2-9 shows the VEA register set. Note that the following programming model is similar to that found in Figure 2-1, however, the VEA registers are now included.

The VEA introduces the time base facility (TB), a 64-bit structure that consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers can be accessed by both user- and supervisor-level instructions. In the context of the VEA, user-level applications are permitted read-only access to the TB. The OEA defines supervisor-level access to the TB for writing values to the TB. See Section 2.3.12, “Time Base Facility (TB)—OEA,” for more information.

In Figure 2-9, the numbers to the right of the register name indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR 1).

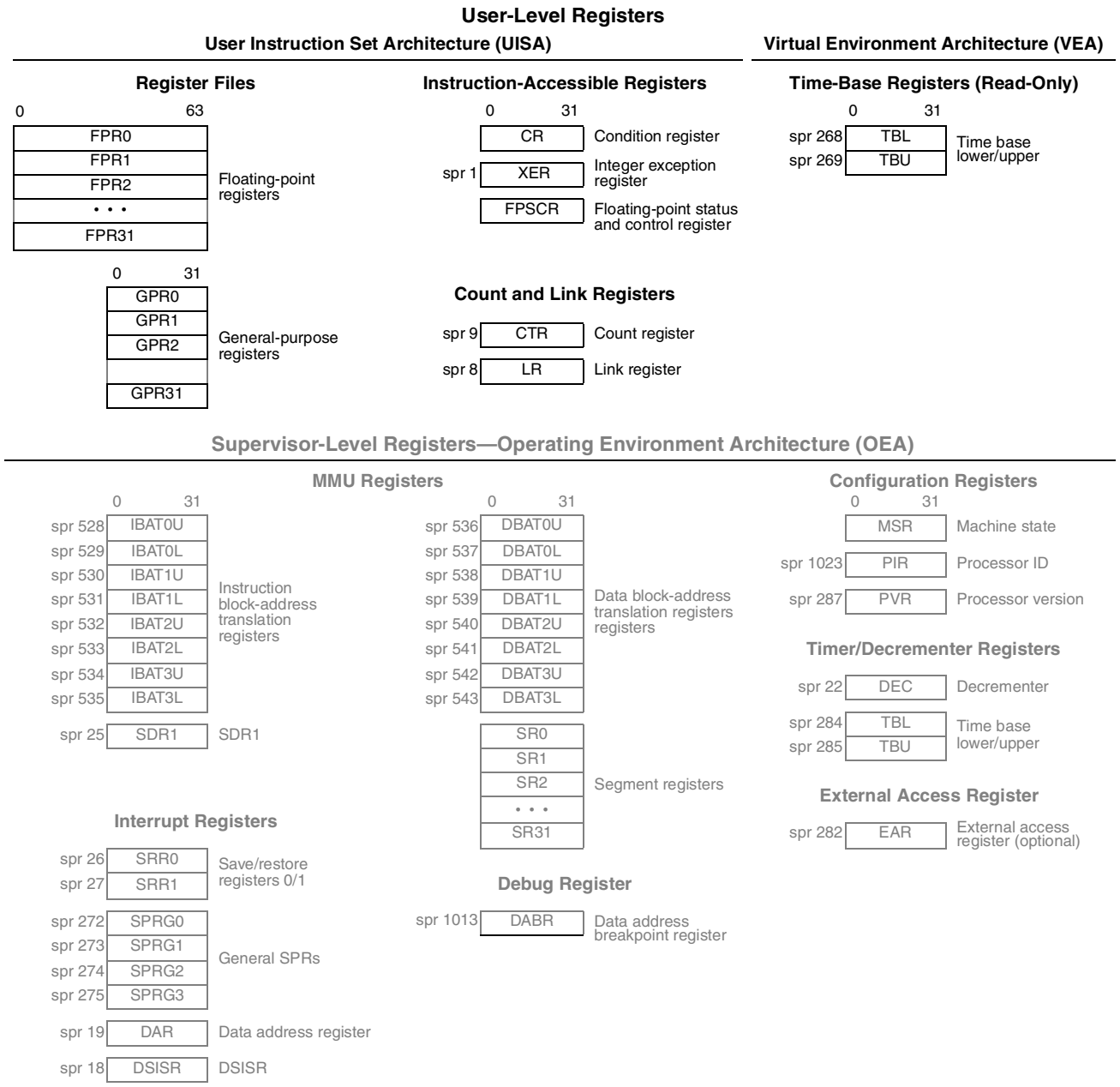


Figure 2-9. VEA Programming Model—User-Level Registers Plus Time Base

The time base (TB), shown in Figure 2-10, is a 64-bit structure that contains a 64-bit unsigned integer that is incremented periodically. Each increment adds 1 to the low-order bit (bit 31 of TBL). The frequency at which the counter is incremented is implementation-dependent.

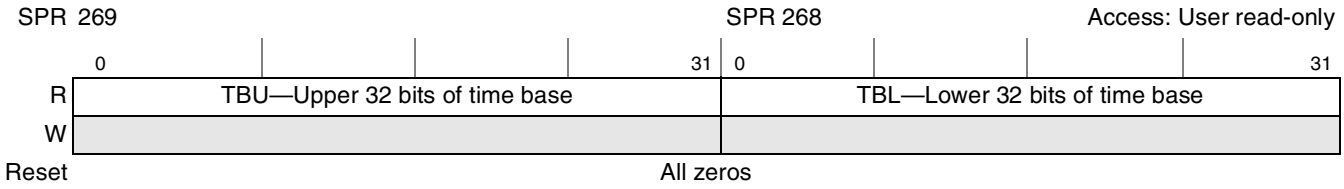


Figure 2-10. Time Base (TB)

The TB increments until its value becomes 0xFFFF_FFFF_FFFF_FFFF ($2^{64} - 1$). At the next increment its value becomes 0x0000_0000_0000_0000. Note that there is no explicit indication that this has occurred (that is, no interrupt is generated).

The period of the time base depends on the driving frequency. The TB is implemented such that the following requirements are satisfied:

1. Loading a GPR from the time base has no effect on the accuracy of the time base.
2. Storing a GPR to the time base replaces the value in the time base with the value in the GPR.

The VEA does not specify a relationship between the frequency at which the time base is updated and other frequencies, such as the processor clock. The TB update frequency is not required to be constant; however, for the system software to maintain time of day and operate interval timers, one of two things is required:

- The system provides an implementation-dependent interrupt to software whenever the update frequency of the time base changes and a means to determine the current update frequency; or
- The system software controls the update frequency of the time base.

Note that if the operating system initializes the TB to some reasonable value and the update frequency of the TB is constant, the TB can be used as a source of values that increase at a constant rate, such as for time stamps in trace entries.

Even if the update frequency is not constant, values read from the TB are monotonically increasing (except when the TB wraps from $2^{64} - 1$ to 0). If a trace entry is recorded each time the update frequency changes, the sequence of TB values can be postprocessed to become actual time values.

However, successive readings of the time base may return identical values due to implementation-dependent factors such as a low update frequency or initialization.

2.2.1 Reading the Time Base

The **mftb** instruction is used to read the time base. For specific details on using the **mftb** instruction, see Chapter 8, “Instruction Set.” For information on writing the time base, see Section 2.3.12.1, “Writing to the Time Base.”

On 32-bit implementations, it is not possible to read the entire 64-bit time base in a single instruction. The **mftb** simplified mnemonic moves from the lower half of the time base register (TBL) to a GPR, and the **mftbu** simplified mnemonic moves from the upper half of the time base (TBU) to a GPR.

Because of the possibility of a carry from TBL to TBU occurring between reads of the TBL and TBU, a sequence such as the following example is necessary to read the time base:

```
loop:
        mftbu   rx      #load from TBU
        mftb    ry      #load from TBL
        mftbu   rz      #load from TBU
        cmpw    rz,rx   #see if 'old' = 'new'
        bne     loop    #loop if carry occurred
```

The comparison and loop are necessary to ensure that a consistent pair of values has been obtained.

2.2.2 Computing Time of Day from the Time Base

Since the update frequency of the time base is system-dependent, the algorithm for converting the current value in the time base to time of day is also system-dependent.

In a system in which the update frequency of the time base may change over time, it is not possible to convert an isolated time base value into time of day. Instead, a time base value has meaning only with respect to the current update frequency and the time of day that the update frequency was last changed. Each time the update frequency changes, either the system software is notified of the change through an interrupt, or else the change was instigated by the system software itself. At each such change, the system software must compute the current time of day using the old update frequency, compute a new value of ticks-per-second for the new frequency, and save the time of day, time base value, and tick rate. Subsequent calls to compute time of day use the current time base value and the saved data.

A generalized service to compute time of day could take the following as input:

- Time of day at beginning of current epoch
- Time base value at beginning of current epoch
- Time base update frequency
- Time base value for which time of day is desired

For a system in which the time base update frequency does not vary, the first three inputs would be constant.

2.3 OEA Register Set

The operating environment architecture (OEA) completes the register model. Figure 2-11. shows the entire register set—UISA, VEA, and OEA. In Figure 2-11 the numbers to the right of the register name indicates the number that is used in the syntax of the instruction operands to access the register (for example, the number used to access the XER is SPR 1).

OEA-defined SPRs can be accessed only by supervisor-level instructions; any attempt to access these SPRs with user-level instructions results in a supervisor-level interrupt. Some SPRs are implementation-specific. In some cases, not all of a register's bits are implemented in hardware.

If a processor executes an **mtspr**/**mfspr** instruction with an undefined SPR encoding, it takes (depending on the implementation) an illegal instruction program interrupt, a privileged instruction program interrupt, or the results are boundedly undefined. See 6.5.7, "Program Interrupt (0x00700)," for more information.

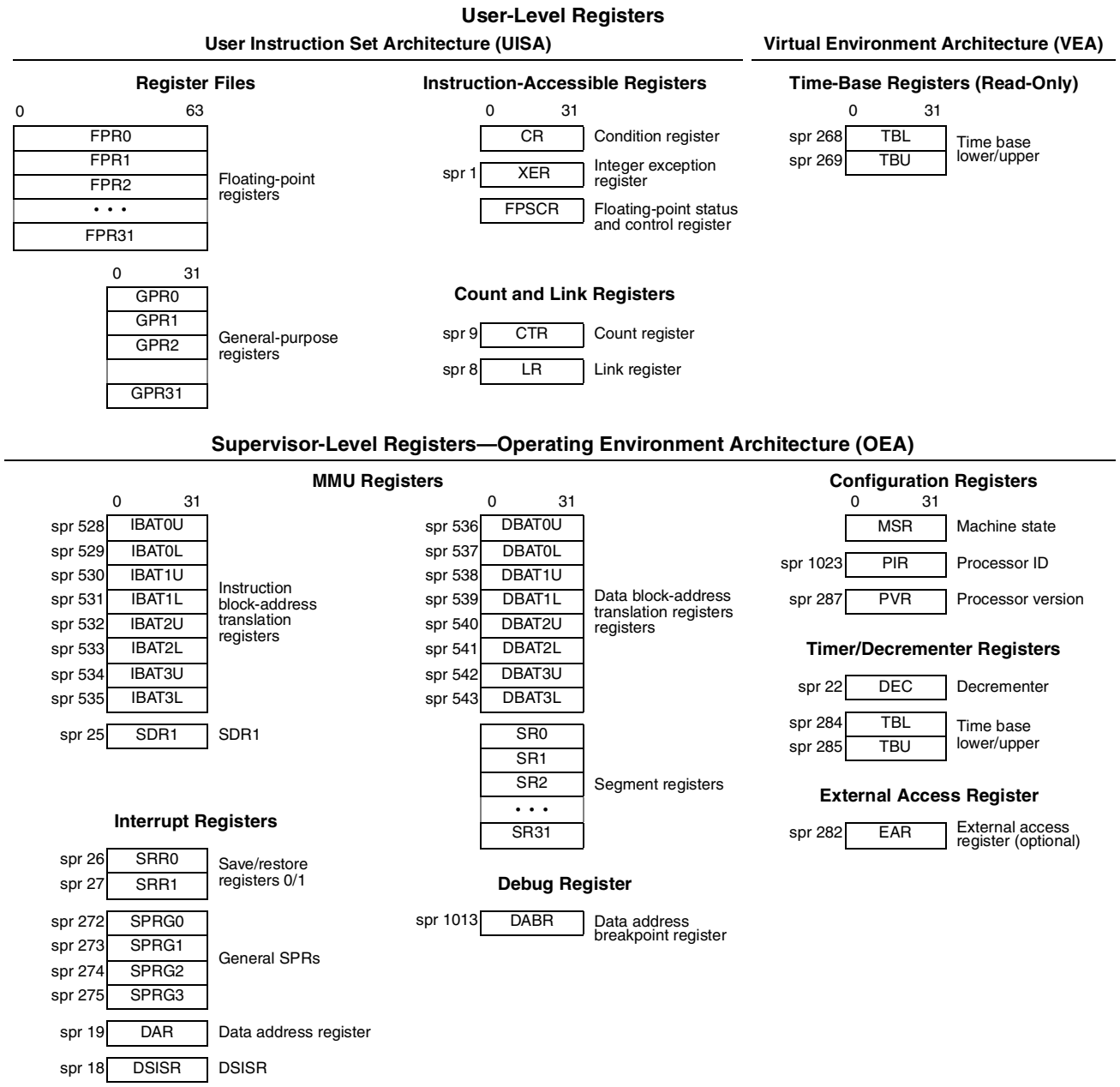


Figure 2-11. OEA Programming Model—All Registers

A description of the OEA supervisor-level registers follows:

- Configuration registers
  - Machine state register (MSR). The MSR defines the state of the processor. The MSR can be modified by the Move to Machine State Register (**mtmsr**), System Call (**sc**), and Return from Interrupt (**rfi**) instructions. It can be read by the Move from Machine State Register (**mfmsr**) instruction. For more information, see Section 2.3.1, “Machine State Register (MSR).”
  - Processor version register (PVR). This register is a read-only register that identifies the version (model) and revision level of the processor. For more information, see Section 2.3.2, “Processor Version Register (PVR).”
- Memory management registers
  - Block-address translation (BAT) registers. The OEA defines four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and four pairs of data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L). Figure 2-11 shows SPR numbers for BAT registers. See Section 2.3.3, “BAT Registers,” for more information.
  - SDR1. The SDR1 register specifies the page table base address used in virtual-to-physical address translation. For more information, see Section 2.3.4, “SDR1.” (Note that physical address is referred to as real address in the architecture specification.)
  - Segment registers (SR). The OEA defines sixteen 32-bit segment registers (SR0–SR15). Note that the SRs are implemented on 32-bit implementations only. The fields in the segment register are interpreted differently depending on the value of bit 0. For more information, see Section 2.3.5, “Segment Registers.”
- Interrupt handling registers
  - Data address register (DAR). After a DSI or an alignment interrupt, DAR is set to the effective address generated by the faulting instruction. For more information, see Section 2.3.6, “Data Address Register (DAR).”
  - SPRG0–SPRG3. The SPRG0–SPRG3 registers are provided for operating system use. For more information, see Section 2.3.7, “SPRG0–SPRG3.”
  - DSISR. The DSISR defines the cause of DSI and alignment interrupts. For more information, refer to Section 2.3.8, “DSISR.”
  - Machine status save/restore register 0 (SRR0). The SRR0 register is used to save machine status on interrupts and to restore machine status when an **rfi** instruction is executed. For more information, see Section 2.3.9, “Machine Status Save/Restore Register 0 (SRR0).”
  - Machine status save/restore register 1 (SRR1). The SRR1 register is used to save machine status on interrupts and to restore machine status when an **rfi** instruction is executed. For more information, see Section 2.3.10, “Machine Status Save/Restore Register 1 (SRR1).”
  - Floating-point exception cause register (FPECR). This optional register is used to identify the cause of a floating-point exception.
- Miscellaneous registers
  - Time base (TB). The TB is a 64-bit structure that maintains the time of day and operates interval timers. The TB consists of two 32-bit registers—time base upper (TBU) and time base lower (TBL). Note that the time base registers can be accessed by both user- and

supervisor-level instructions. For more information, see Section 2.3.12, “Time Base Facility (TB)—OEA,” and Section 2.2, “VEA Register Set—Time Base.”

— Decrementer register (DEC). This register is a 32-bit decrementing counter that provides a mechanism for causing a decrementer interrupt after a programmable delay; the frequency is a subdivision of the processor clock. For more information, see Section 2.3.13, “Decrementer Register (DEC).”

— External access register (EAR). This optional register is used in conjunction with the **eciwx** and **ecowx** instructions. Note that the EAR register and the **eciwx** and **ecowx** instructions are optional in the architecture and may not be supported in all processors that implement the OEA. For more information about the external control facility, see Section 4.3.4, “External Control Instructions (Optional).”

— Data address breakpoint register (DABR). This optional register is used to control the data address breakpoint facility. Note that the DABR is optional in the architecture and may not be supported in all processors that implement the OEA. For more information about the data address breakpoint facility, see Section 6.5.3, “Data Storage Interrupt (0x00300).”

— Processor identification register (PIR). This optional register is used to hold a value that distinguishes an individual processor in a multiprocessor environment.

2.3.1 Machine State Register (MSR)

The machine state register (MSR), shown in Figure 2-12, defines the state of the processor. When an interrupt occurs, MSR bits, as described in Table 2-8, are altered as determined by the interrupt. The MSR can also be modified by the **mtmsr**, **sc**, and **rfi** instructions. It can be read by the **mfmsr** instruction.

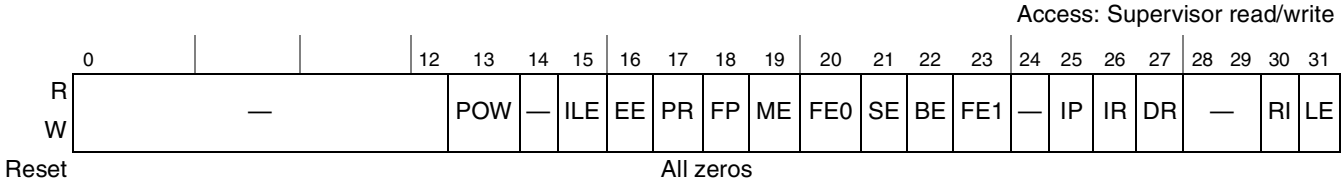


Figure 2-12. Machine State Register (MSR)

Table 2-9 shows the bit definitions for the MSR.

Table 2-9. MSR Bit Settings

Bits	Name	Description
0–12	—	Reserved
13	POW	Power management enable 0 Power management disabled (normal operation mode) 1 Power management enabled (reduced power mode) Note: Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved.
14	—	Reserved
15	ILE	Interrupt little-endian mode. When an interrupt occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the interrupt.

Table 2-9. MSR Bit Settings (continued)

Bits	Name	Description
16	EE	External interrupt enable 0 While the bit is cleared, the processor delays recognition of external interrupts and decrementer interrupt conditions. 1 The processor is enabled to take an external interrupt or the decrementer interrupt.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
19	ME	Machine check enable 0 Machine check interrupts are disabled. 1 Machine check interrupts are enabled.
20	FE0	Floating-point exception mode 0 (see Table 2-10).
21	SE	Single-step trace enable (Optional) 0 The processor executes instructions normally. 1 The processor generates a single-step trace interrupt upon the successful execution of the next instruction. Note: If the function is not implemented, this bit is treated as reserved.
22	BE	Branch trace enable (Optional) 0 The processor executes branch instructions normally. 1 The processor generates a branch trace interrupt after completing the execution of a branch instruction, regardless of whether the branch was taken. Note: If the function is not implemented, this bit is treated as reserved.
23	FE1	Floating-point exception mode 1 (See Table 2-10).
24	—	Reserved
25	IP	Interrupt prefix. The setting of this bit specifies whether an interrupt vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the interrupt vector. See Table 6-2. 0 Interrupts are vectored to the physical address 0x000*n_nnnn*. 1 Interrupts are vectored to the physical address 0xFFF*n_nnnn*. In most systems, IP is set during system initialization and then cleared when initialization is complete.
26	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information, see Chapter 7, “Memory Management.”
27	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information, see Chapter 7, “Memory Management.”
28–29	—	Reserved

Table 2-9. MSR Bit Settings (continued)

Bits	Name	Description
30	RI	Recoverable interrupt (for system reset and machine check interrupts). 0 Interrupt is not recoverable. 1 Interrupt is recoverable. For more information, see Chapter 6, “Interrupts.”
31	LE	Little-endian mode enable 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

The floating-point exception mode bits (FE0–FE1) are interpreted as shown in Table 2-10.

Table 2-10. Floating-Point Interrupt Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions disabled
0	1	Floating-point imprecise nonrecoverable
1	0	Floating-point imprecise recoverable
1	1	Floating-point precise mode

Table 2-11 indicates the initial state of the MSR at power up.

Table 2-11. State of MSR at Power Up

Bits	Name	32-Bit Default Value
0–12	—	Unspecified[1]
13	POW	0
14	—	Unspecified[1]
15	ILE	0
16	EE	0
17	PR	0
18	FP	0
19	ME	0
20	FE0	0
21	SE	0
22	BE	0
23	FE1	0
24	—	Unspecified[1]
25	IP	1[2]
26	IR	0
27	DR	0

Table 2-11. State of MSR at Power Up (continued)

Bits	Name	32-Bit Default Value
28–29	—	Unspecified [1]
30	RI	0
31	LE	0

[1] Unspecified can be either 0 or 1

[2] 1 is typical, but might be 0

2.3.2 Processor Version Register (PVR)

The processor version register (PVR) is a 32-bit, read-only register that contains a value identifying the specific version (model) and revision level of the processor. The contents of the PVR can be copied to a GPR by the **mfspr** instruction. Read access to the PVR is supervisor-level only; write access is not provided.

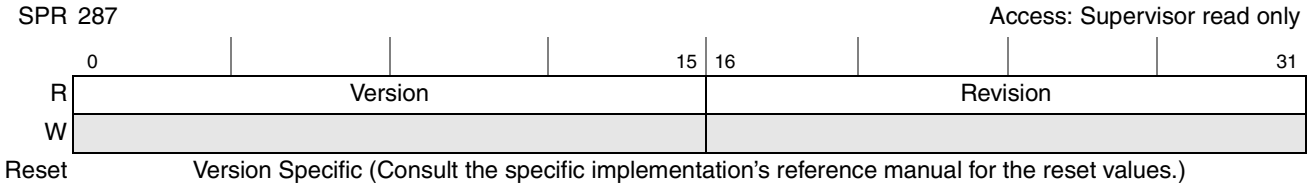


Figure 2-13. Processor Version Register (PVR)

The PVR consists of two 16-bit fields, described in Table 2-12.

Table 2-12. PVR Field Descriptions

Bits	Name	Description
0–15	Version	A 16-bit number that uniquely identifies a particular processor version. This number can be used to determine the version of a processor; it may not distinguish between different end product models if more than one model uses the same processor.
16–31	Revision	A 16-bit number that distinguishes between various releases of a particular version (that is, an engineering change level). The value of the revision portion of the PVR is implementation-specific. The processor revision level is changed for each revision of the device.

2.3.3 BAT Registers

The BAT registers (BATs) maintain the address translation information for eight blocks of memory. The BATs are maintained by the system software and are implemented as eight pairs of special-purpose registers (SPRs). Each block is defined by a pair of SPRs called upper and lower BAT registers. These BAT registers define the starting addresses and sizes of BAT areas.

The OEA defines eight instruction block-address translation (IBAT) registers, consisting of four pairs of instruction BATs (IBAT0U–IBAT3U and IBAT0L–IBAT3L) and eight data BATs (DBAT0U–DBAT3U and DBAT0L–DBAT3L).

Figure 2-14 shows the format of the upper and lower BATs for 32-bit processors.

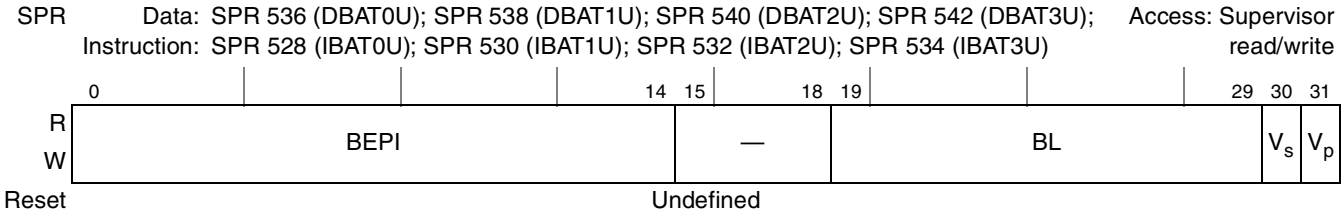
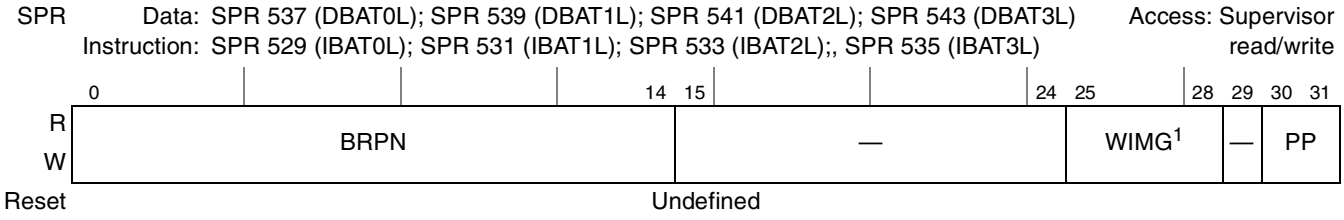


Figure 2-14. Format of Upper BAT Register

Figure 2-15 shows the format of the lower BATs for 32-bit processors.



1 W and G bits are not defined for IBAT registers. Attempting to write to these bits causes boundedly-undefined results.

Figure 2-15. Format of Lower BAT Register

Table 2-13 describes the bits in the BAT registers.

Table 2-14. BAT Registers—Field and Bit Descriptions

Upper/Lower BAT	Bits	Name	Description
Upper BAT Register	0–14	BEPI	Block effective page index. This field is compared with high-order bits of the logical address to determine if there is a hit in that BAT array entry. (Note that the architecture specification refers to logical address as effective address.)
	15–18	—	Reserved
	19–29	BL	Block length. BL is a mask that encodes the size of the block. Values for this field are listed in Table 2-15. See 7.5.3, “BAT Register Implementation of BAT Array.”
	30	Vs	Supervisor mode valid bit. This bit interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.5.2, “Recognition of Addresses in BAT Arrays."
	31	Vp	User mode valid bit. This bit also interacts with MSR[PR] to determine if there is a match with the logical address. For more information, see Section 7.5.2, “Recognition of Addresses in BAT Arrays.”

Table 2-14. BAT Registers—Field and Bit Descriptions (continued)

Upper/Lower BAT	Bits	Name	Description
Lower BAT Register	0–14	BRPN	This field is used in conjunction with the BL field to generate high-order bits of the physical address of the block.
	15–24	—	Reserved
	25–28	WIMG	Memory/cache access mode bits W Write-through I Caching-inhibited M Memory coherence G Guarded Attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results. For detailed information about the WIMG bits, see Section 5.3.1, “Memory/Cache Access Attributes."
	29	—	Reserved
	30–31	PP	Protection bits for block. This field determines the protection for the block as described in Section 7.5.4, “Block Memory Protection."

Table 2-15 lists the BAT area lengths encoded in BAT[BL].

Table 2-15. BAT Area Lengths

BAT Area Length	BL Encoding
128 Kbytes	000 0000 0000
256 Kbytes	000 0000 0001
512 Kbytes	000 0000 0011
1 Mbyte	000 0000 0111
2 Mbytes	000 0000 1111
4 Mbytes	000 0001 1111
8 Mbytes	000 0011 1111
16 Mbytes	000 0111 1111
32 Mbytes	000 1111 1111
64 Mbytes	001 1111 1111
128 Mbytes	011 1111 1111
256 Mbytes	111 1111 1111

Only the values in Table 2-15 are valid for the BL field. The rightmost BL bit is aligned with bit 14 of the logical address. A logical address is determined to be within a BAT area if the logical address matches the value in the BEPI field.

The boundary between the cleared bits and set bits in BL determines the bits of logical address that participate in the comparison with BEPI. Bits in the logical address corresponding to set bits in BL are cleared for this comparison. Bits in the logical address corresponding to set bits in the BL field,

concatenated with the 17 bits of the logical address to the right (less significant bits) of BL, form the offset within the BAT area. This is described in detail in Section 7.5.3, “BAT Register Implementation of BAT Array.”

The value loaded into BL determines both the length of the BAT area and the alignment of the area in both logical and physical address space. The values loaded into BEPI and BRPN must have at least as many low-order zeros as there are ones in BL.

Use of BAT registers is described in Section 7.5, “Block Address Translation.”

2.3.4 SDR1

The 32-bit implementation of SDR1 is shown in Figure 2-16.

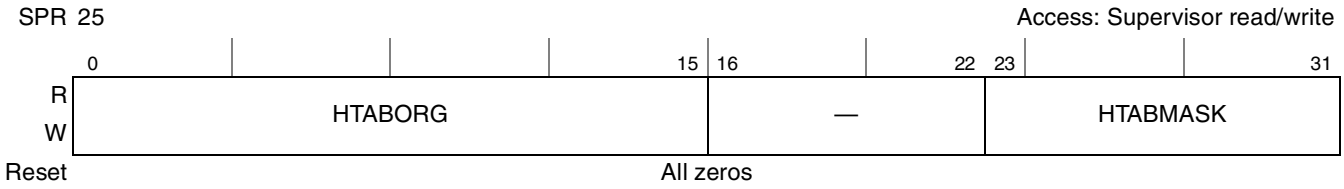


Figure 2-16. SDR1 Register Format

The SDR1 fields are described in Table 2-16.

Table 2-16. SDR1 Bit Settings

Bits	Name	Description
0–15	HTABORG	The high-order 16 bits of the 32-bit physical address of the page table
16–22	—	Reserved
23–31	HTABMASK	Mask for page table address

In 32-bit implementations, the HTABORG field in SDR1 contains the high-order 16 bits of the 32-bit physical address of the page table. Therefore, the page table is constrained to lie on a 2^{16}-byte (64 Kbytes) boundary at a minimum. At least 10 bits from the hash function are used to index into the page table. The page table must consist of at least 64 Kbytes (2^{10} PTEGs of 64 bytes each).

The page table can be any size 2^n where $16 \le n \le 25$. As the table size is increased, more bits are used from the hash to index into the table and the value in HTABORG must have more of its low-order bits equal to 0. The HTABMASK field in SDR1 contains a mask value that determines how many bits from the hash are used in the page table index. This mask must be of the form 0b00...011...1; that is, a string of 0 bits followed by a string of 1bits. The 1 bits determine how many additional bits (at least 10) from the hash are used in the index; HTABORG must have this same number of low-order bits equal to 0.

For example, suppose that the page table is 8,192 (2^{13}), 64-byte PTEGs, for a total size of 2^{19} bytes (512 Kbytes). Note that a 13-bit index is required. Ten bits are provided from the hash initially, so 3 additional bits form the hash must be selected. The value in HTABMASK must be 0x007 and the value in HTABORG must have its low-order 3 bits (SDR1[13–15]) equal to 0. This means that the page table must begin on a $2^{3 + 10 + 6} = 2^{19} = 512$-Kbyte boundary.

For more information, refer to Chapter 7, “Memory Management.”

2.3.5 Segment Registers

The segment registers contain the segment descriptors for 32-bit implementations. For 32-bit processors, the OEA defines a segment register file of sixteen 32-bit registers. Segment registers can be accessed by using the **mtsr/mfsr** and **mtsrin/mfsrin** instructions. The value of bit 0, the T bit, determines how the remaining register bits are interpreted. Figure 2-17 shows the format of a segment register when T = 0.

Figure 2-17. Segment Register Format (T = 0)

Segment register bit settings when T = 0 are described in Table 2-17.

Table 2-17. Segment Register Bit Settings (T = 0)

Bits	Name	Description
0	T	T = 0 selects this format
1	Ks	Supervisor-state protection key
2	Kp	User-state protection key
3	N	No-execute protection
4–7	—	Reserved
8–31	VSID	Virtual segment ID

Figure 2-18 shows the bit definition when T = 1.

Figure 2-18. Segment Register Format (T = 1)

The bits in the segment register when T = 1 are described in Table 2-18.

Table 2-18. Segment Register Bit Settings (T = 1)

Bits	Name	Description
0	T	T = 1 selects this format.
1	Ks	Supervisor-state protection key
2	Kp	User-state protection key
3–11	BUID	Bus unit ID
12–31	CNTLR_SPEC	Device-specific data for I/O controller

If an access is translated by the block address translation (BAT) mechanism, the BAT translation takes precedence and the results of translation using segment registers are not used. However, if an access is not translated by a BAT, and T = 0 in the selected segment register, the effective address is a reference to a

memory-mapped segment. In this case, the 52-bit virtual address (VA) is formed by concatenating the following:

- The 24-bit VSID field from the segment register
- The 16-bit page index, EA[4–19]
- The 12-bit byte offset, EA[20–31]

The VA is then translated to a physical address as described in Section 7.6, “Memory Segment Model.”

If T = 1 in the selected segment register (and the access is not translated by a BAT), the effective address is a reference to a direct-store segment, defined by the architecture but not supported. No reference is made to the page tables.

2.3.6 Data Address Register (DAR)

The DAR is shown in Figure 2-19.

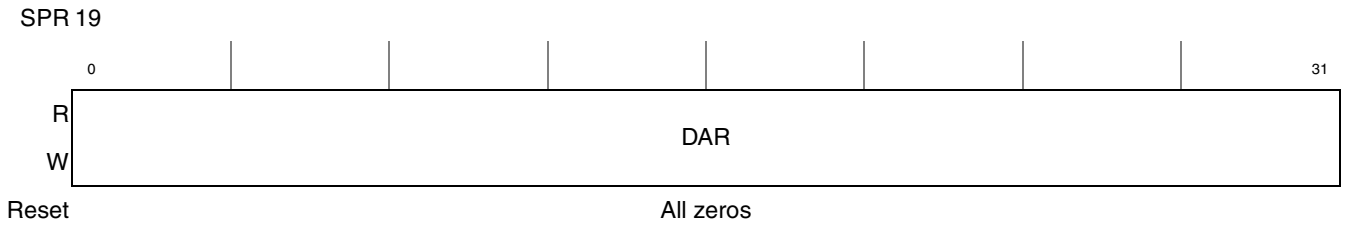


Figure 2-19. Data Address Register (DAR)

The effective address generated by a memory access instruction is placed in the DAR if the access causes an interrupt (for example, an alignment interrupt). For information, see Chapter 6, “Interrupts.”

2.3.7 SPRG0–SPRG3

SPRG0–SPRG3 are provided for general operating system use, such as performing a fast state save or for supporting multiprocessor implementations. The formats of SPRG0–SPRG3 are shown in Figure 2-20.

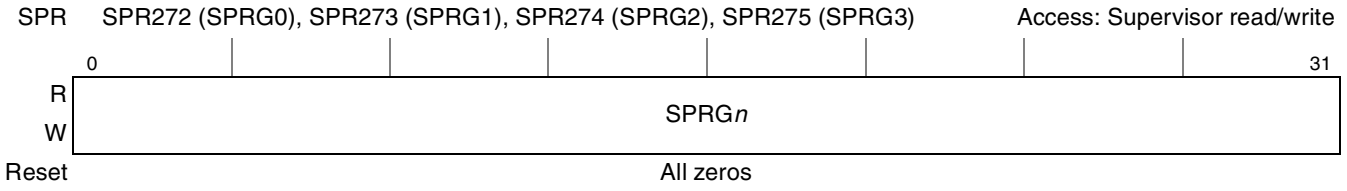


Figure 2-20. SPRG0–SPRG3

Table 2-19 describes typical uses of SPRG0 through SPRG3.

Table 2-19. Conventional Uses of SPRG0–SPRG3

Register	Description
SPRG0	Software may load a unique physical address in this register to identify an area of memory reserved for use by the first-level interrupt handler. This area must be unique for each processor in the system.
SPRG1	SPRG1 may be used as a scratch register by the first-level interrupt handler to save the content of a GPR. That GPR then can be loaded from SPRG0 and used as a base register to save other GPRs to memory.
SPRG2	SPRG2 may be used by the operating system as needed.
SPRG3	SPRG3 may be used by the operating system as needed.

2.3.8 DSISR

The DSISR, shown in Figure 2-21, identifies the cause of DSI and alignment interrupts.

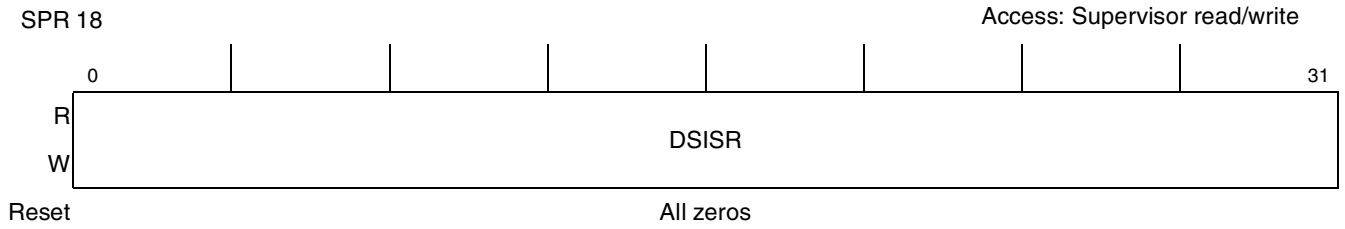


Figure 2-21. DSISR

For information about bit settings, see Section 6.5.3, “Data Storage Interrupt (0x00300),” and Section 6.5.6, “Alignment Interrupt (0x00600).”

2.3.9 Machine Status Save/Restore Register 0 (SRR0)

The SRR0 is used to save machine status on interrupts and restore machine status when an **rfi** instruction is executed. It also holds the EA for the instruction that follows the System Call (**sc**) instruction. The format of SRR0 is shown in Figure 2-22.

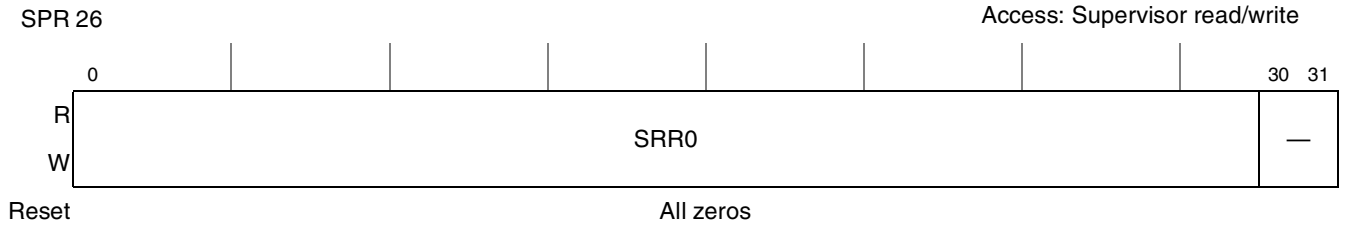


Figure 2-22. Machine Status Save/Restore Register 0 (SRR0)

When an interrupt occurs, SRR0 is set to point to an instruction such that all prior instructions have completed execution and no subsequent instruction has begun execution. When an **rfi** instruction is executed, the contents of SRR0 are copied to the next instruction address (NIA)—the 64- or 32-bit address of the next instruction to be executed. The instruction addressed by SRR0 may not have completed

execution, depending on the interrupt type. SRR0 addresses either the instruction causing the interrupt or the immediately following instruction. The instruction addressed can be determined from the interrupt type and status bits.

Note that in some implementations, every instruction fetch performed while MSR[IR] = 1, and every instruction execution requiring address translation when MSR[DR] = 1, may modify SRR0.

For information on how specific interrupts affect SRR0, refer to the descriptions of individual interrupts in Chapter 6, “Interrupts.”

2.3.10 Machine Status Save/Restore Register 1 (SRR1)

SRR1 is used to save machine status on interrupts and to restore machine status when an **rfi** instruction is executed. Figure 2-23 shows the SRR1 format.

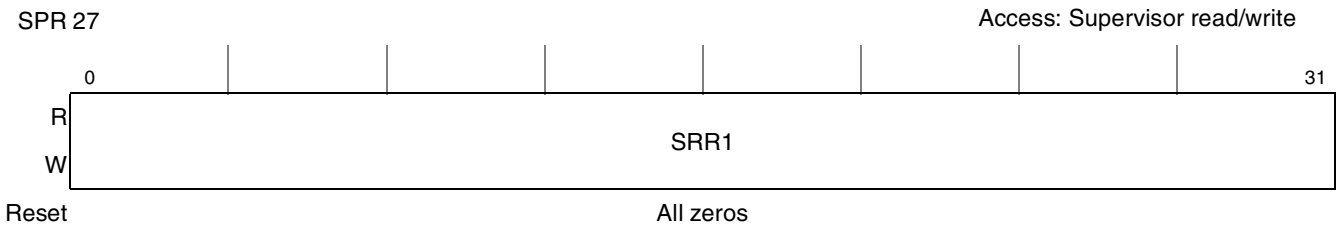


Figure 2-23. Machine Status Save/Restore Register 1 (SRR1)

When an interrupt occurs, SRR1[1–4,10–15] are loaded with interrupt-specific information and MSR[16–23,25–27,30–31] are placed in corresponding SRR1 bit positions. When **rfi** executes, MSR[16–23,25–27,30–31] are loaded from SRR1[16–23,25–27,30–31].

The remaining bits of SRR1 are defined as reserved. An implementation may define one or more of these bits, and in this case, may also cause them to be saved from MSR on an interrupt and restored to MSR from SRR1 on an **rfi**.

Note that, in some implementations, every instruction fetch when MSR[IR] = 1, and every instruction execution requiring address translation when MSR[DR] = 1, may modify SRR1.

For information on how specific interrupts affect SRR1, refer to the individual interrupts in Chapter 6, “Interrupts.”

2.3.11 Floating-Point Exception Cause Register (FPECR)

FPECR may be used to identify the cause of a floating-point exception. Note that the FPECR is an optional register in the architecture and may be implemented differently (or not at all) in the design of each processor. The user’s manual of a specific processor will describe the functionality of the FPECR, if it is implemented in that processor.

2.3.12 Time Base Facility (TB)—OEA

As described in Section 2.2, “VEA Register Set—Time Base,” the time base (TB) provides a long-period counter driven by an implementation-dependent frequency. The VEA defines user-level read-only access to the TB. Writing to the TB is reserved for supervisor-level applications such as operating systems and boot-strap routines. The OEA defines supervisor-level, write access to the TB.

The TB is a volatile resource and must be initialized during reset. Some implementations may initialize the TB with a known value; however, there is no guarantee of automatic initialization of the TB when the processor is reset. The TB runs continuously at start-up.

For more information on the user-level aspects of the time base, refer to Section 2.2, “VEA Register Set—Time Base.”

2.3.12.1 Writing to the Time Base

Note that writing to the TB is reserved for supervisor-level software.

The simplified mnemonics, **mttbl** and **mttbu**, write the lower and upper halves of the TB, respectively. The simplified mnemonics listed above are for the **mtspr** instruction; see Appendix E, “Simplified Mnemonics for PowerPC Instructions.” The **mtspr**, **mttbl**, and **mttbu** instructions treat TBL and TBU as separate 32-bit registers; setting one leaves the other unchanged. It is not possible to write the entire 64-bit time base in a single instruction.

The TB can be written by a sequence such as the following:

```
lwz     rx,upper                #load 64-bit value for
lwz     ry,lower                # TB into rx and ry
li      rz,0
mttbl   rz                      #force TBL to 0
mttbu   rx                      #set TBU
mttbl   ry                      #set TBL
```

Provided that no interrupts occur while the last three instructions are being executed, loading 0 into TBL prevents the possibility of a carry from TBL to TBU while the time base is being initialized.

For information on reading the time base, refer to Section 2.2.1, “Reading the Time Base.”

2.3.13 Decrementer Register (DEC)

The decrementer register (DEC), shown in Figure 2-24, is a 32-bit decrementing counter that provides a mechanism for causing a decrementer interrupt after a programmable delay. The DEC frequency is based on the same implementation-dependent frequency that drives the time base.

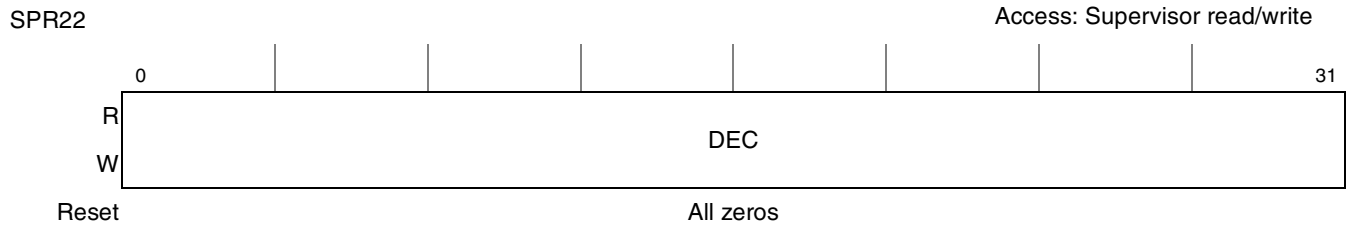


Figure 2-24. Decrementer Register (DEC)

2.3.13.1 Decrementer Operation

The DEC counts down, causing an interrupt (unless masked by MSR[EE]) when it passes through zero. The DEC satisfies the following requirements:

- The operation of the time base and the DEC are coherent (that is, the counters are driven by the same fundamental time base).
- Loading a GPR from the DEC has no effect on the DEC.
- Storing the contents of a GPR to the DEC replaces the value in the DEC with the value in the GPR.
- Whenever bit 0 of the DEC changes from 0 to 1, a decrementer interrupt request is signaled. Multiple DEC interrupt requests may be received before the first interrupt occurs; however, any additional requests are canceled when the interrupt occurs for the first request.
- If the DEC is altered by software and the content of bit 0 is changed from 0 to 1, an interrupt request is signaled.

2.3.13.2 Writing and Reading the DEC

The content of the DEC can be read or written using the **mfspr** and **mtspr** instructions, both of which are supervisor-level when they refer to the DEC. Using a simplified mnemonic for the **mtspr** instruction, the DEC may be written from GPR **r**A with the following:

```
mtdec    rA
```

Likewise, the DEC may be read into GPR **r**A with the following simplified mnemonic:

```
mfdec    rA
```

2.3.14 Data Address Breakpoint Register (DABR)

The optional data address breakpoint facility is controlled by an optional SPR, the DABR. The data address breakpoint facility is optional to the architecture. However, if the data address breakpoint facility is implemented, it is recommended, but not required, that it be implemented as described in this section.

The data address breakpoint facility provides a means to detect accesses to a designated double word. The address comparison is done on an effective address, and it applies to data accesses only. It does not apply to instruction fetches.

The DABR is shown8 in Figure 2-25.

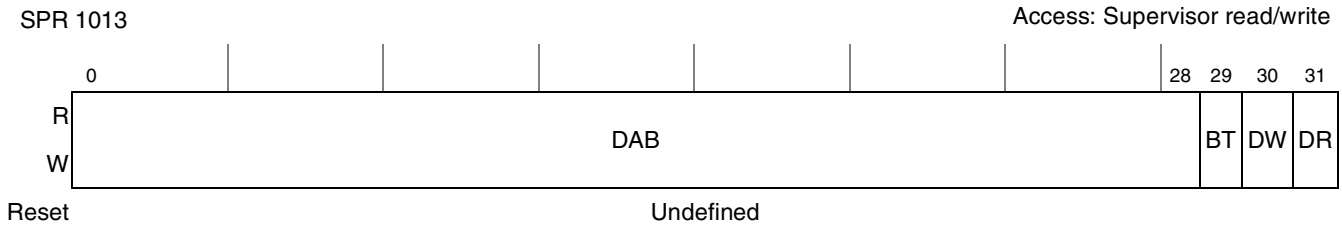


Figure 2-25. Data Address Breakpoint Register (DABR)

Table 2-20 describes the fields in the DABR.

Table 2-20. DABR—Bit Settings

Bits	Name	Description
0–28	DAB	Data address breakpoint
29	BT	Breakpoint translation enable
30	DW	Data write enable
31	DR	Data read enable

A data address breakpoint match is detected for a load or store instruction if the three following conditions are met for any byte accessed:

- EA[0–28] = DABR[DAB]
- MSR[DR] = DABR[BT]
- The instruction is a store and DABR[DW] = 1, or the instruction is a load and DABR[DR] = 1.

Even if the above conditions are satisfied, it is undefined whether a match occurs in the following cases:

- A store string instruction (**stwcx.**) in which the store is not performed
- A load or store string instruction (**lswx** or **stswx**) with a zero length
- A **dcbz**, **dcba**, **eciwx**, or **ecowx** instruction. For the purpose of determining whether a match occurs, **eciwx** is treated as a load, and **dcbz**, **dcba**, and **ecowx** are treated as stores.

The cache management instructions other than **dcbz** and **dcba** never cause a match. If **dcbz** or **dcba** causes a match, some or all of the target memory locations may have been updated.

A match generates a DSI interrupt. Section 6.5.3, “Data Storage Interrupt (0x00300),” gives more information on the data address breakpoint facility.

2.3.15 External Access Register (EAR)

The EAR is an optional 32-bit SPR that controls access to the external control facility and identifies the target device for external control operations. The external control facility provides a means for user-level instructions to communicate with special external devices. The EAR is shown in Figure 2-26.

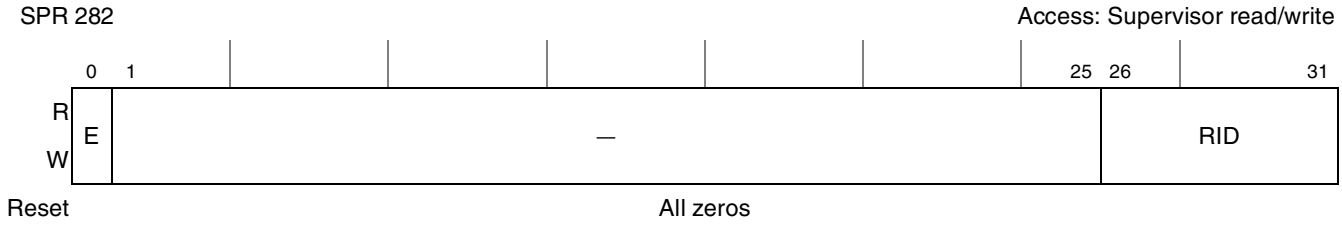


Figure 2-26. External Access Register (EAR)

The high-order bits of the resource ID (RID) field beyond the width of the RID supported by a particular implementation are treated as reserved bits.

The EAR register is provided to support the External Control In Word Indexed (**eciwx**) and External Control Out Word Indexed (**ecowx**) instructions, which are described in Chapter 8, "Instruction Set." Although access to the EAR is supervisor-level, the operating system can determine which tasks are allowed to issue external access instructions and when they are allowed to do so. EAR bit settings described in Table 2-21. Interpretation of the physical address transmitted by the **eciwx** and **ecowx** instructions and the 32-bit value transmitted by the **ecowx** instruction is not prescribed by the OEA but is determined by the target device. The data access of **eciwx** and **ecowx** is performed as though the memory access mode bits (WIMG) were 0101.

For example, if the external control facility supports a graphics adapter, **ecowx** could be used to send the translated physical address of a buffer containing graphics data to the graphics device; **eciwx** could be used to load status information from the graphics adapter.

Table 2-21. External Access Register (EAR) Bit Settings

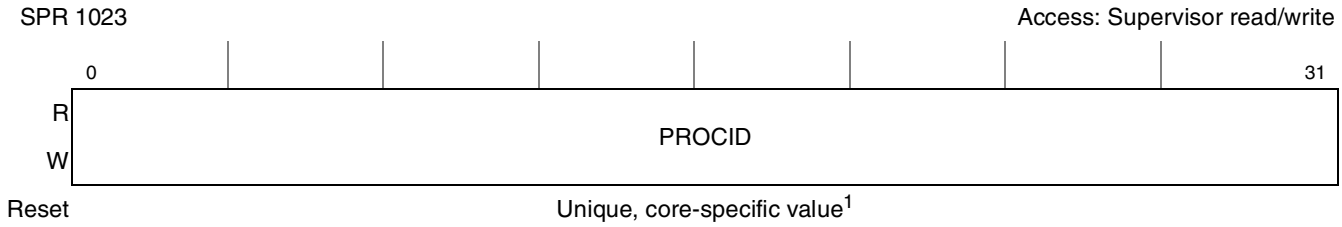
Bits	Name	Description
0	E	Enable bit 0 Disabled. **eciwx** or **ecowx** causes a DSI interrupt. 1 Enabled. **eciwx** and **ecowx** can perform the specified external operation.
1–25	—	Reserved, should be cleared.
26–31	RID	Resource ID

EAR can be accessed by using the **mtspr** and **mfspr**. Table 2-23 and Table 2-24 show EAR synchronization requirements.

2.3.16 Processor Identification Register (PIR)

The optional, 32-bit processor identification register (PIR) is a read-only register that contains a value that can be used to distinguish the processor from other processors in the system. The contents of the PIR can be copied to a GPR by the **mfspr** instruction.

Read access to the PIR is privileged; write access, if provided, is implementation dependent.



1 Consult specific implementation's reference manual.

Figure 2-27. Processor Identification Register (PIR)

Table 2-22. PID Field Description

Bits	Name	Description
0–31	PROCID	Processor ID

2.3.17 Synchronization Requirements for Special Registers and for Lookaside Buffers

Changing the value in certain system registers, and invalidating TLB entries, can cause alteration of the context in which data addresses and instruction addresses are interpreted, and in which instructions are executed. An instruction that alters the context in which data addresses or instruction addresses are interpreted, or in which instructions are executed, is called a context-altering instruction. The context synchronization required for context-altering instructions is shown in Table 2-23 for data access and Table 2-24 for instruction fetch and execution.

A context-synchronizing interrupt (that is, any interrupt except nonrecoverable system reset or nonrecoverable machine check) can be used instead of a context-synchronizing instruction. In the tables, if no software synchronization is required before (after) a context-altering instruction, the synchronizing instruction before (after) the context-altering instruction should be interpreted as meaning the context-altering instruction itself.

A synchronizing instruction before the context-altering instruction ensures that all instructions up to and including that synchronizing instruction are fetched and executed in the context that existed before the alteration. A synchronizing instruction after the context-altering instruction ensures that all instructions after that synchronizing instruction are fetched and executed in the context established by the alteration. Instructions after the first synchronizing instruction, up to and including the second synchronizing instruction, may be fetched or executed in either context.

If an instruction sequence alters the context but contains no instructions affected the alterations, no software synchronization is required within the sequence.

Note that some instructions that occur naturally in the program, such as the **rfi** at the end of an interrupt handler, provide the required synchronization.

No software synchronization is needed before altering the MSR (except when altering MSR[POW] or MSR[LE]; see Table 2-23 and Table 2-24), because **mtmsr** is execution synchronizing. No software

synchronization is required before most of the other alterations shown in Table 2-24, because instructions before the context-altering instruction are fetched and decoded before the context-altering instruction is executed (the processor must determine whether any of the preceding instructions are context synchronizing). Table 2-23 provides information on data access synchronization requirements.

Table 2-23. Data Access Synchronization

Instruction/Event	Required Prior	Required After
Interrupt [1]	None	None
rfi [1]	None	None
sc [1]	None	None
Trap [1]	None	None
mtmsr (ILE)	None	None
mtmsr (PR)	None	Context-synchronizing instruction
mtmsr (ME) [2]	None	Context-synchronizing instruction
mtmsr (DR)	None	Context-synchronizing instruction
mtmsr (LE) [3]	—	—
mtsr [or **mtsrin**]	Context-synchronizing instruction	Context-synchronizing instruction
mtspr (SDR1) [4, 5]	**sync**	Context-synchronizing instruction
mtspr (DBAT)	Context-synchronizing instruction	Context-synchronizing instruction
mtspr (DABR) [6]	—	—
mtspr (EAR)	Context-synchronizing instruction	Context-synchronizing instruction
tlbie [7, 8]	Context-synchronizing instruction	Context-synchronizing instruction or **sync**
tlbia [7, 8]	Context-synchronizing instruction	Context-synchronizing instruction or **sync**

1 Synchronization requirements for changing the power conserving mode are implementation-dependent.

2 A context synchronizing instruction is required after modification of the MSR[ME] bit to ensure that the modification takes effect for subsequent machine check interrupts, which may not be recoverable and therefore may not be context synchronizing.

3 Synchronization requirements for changing from one endian mode to the other are implementation-dependent.

4 SDR1 must not be altered when MSR[DR] = 1 or MSR[IR] = 1; if it is, the results are undefined.

5 A **sync** instruction is required before the **mtspr** instruction because SDR1 identifies the page table and thereby the location of the reference and change (R and C) bits. To ensure that R and C bits are updated in the correct page table, SDR1 must not be altered until all R and C bit updates due to instructions before the **mtspr** have completed. A **sync** instruction guarantees this synchronization of R and C bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

6 Synchronization requirements for changing the DABR are implementation-dependent.

7 For data accesses, the context synchronizing instruction before the **tlbie**, or **tlbia** instruction ensures that all memory accesses, due to preceding instructions, have completed to a point at which they have reported all interrupts that may be caused. The context synchronizing instruction after the **tlbie**, or **tlbia** ensures that subsequent memory accesses will not use the TLB entries being invalidated. It does not ensure that all memory accesses previously translated by the TLB entries being invalidated have completed with respect to memory or, for **tlbie** or **tlbia**, that R and C bit updates associated with those memory accesses have completed; if these completions must be ensured, the **tlbie**, or **tlbia** must be followed by a **sync** instruction rather than by a context synchronizing instruction.

8 Multiprocessor systems have other requirements to synchronize TLB invalidate.

For information on instruction access synchronization requirements, see Table 2-24.

Table 2-24. Instruction Access Synchronization

Instruction/Event	Required Prior	Required After
Interrupt [1]	None	None
rfi [1]	None	None
sc [1]	None	None
Trap [1]	None	None
mtmsr (POW) [1]	—	—
mtmsr (ILE)	None	None
mtmsr (EE) [2]	None	None
mtmsr (PR)	None	Context-synchronizing instruction
mtmsr (FP)	None	Context-synchronizing instruction
mtmsr (ME) [3]	None	Context-synchronizing instruction
mtmsr (FE0, FE1)	None	Context-synchronizing instruction
mtmsr (SE, BE)	None	Context-synchronizing instruction
mtmsr (IP)	None	None
mtmsr (IR) [4]	None	Context-synchronizing instruction
mtmsr (RI)	None	None
mtmsr (LE) [5]	—	—
mtsr [or **mtsrin**] [4]	None	Context-synchronizing instruction
mtspr (SDR1) [6, 7]	**sync**	Context-synchronizing instruction
mtspr (IBAT) [4]	None	Context-synchronizing instruction
mtspr (DEC) [8]	None	None
tlbie [9, 10]	None	Context-synchronizing instruction or **sync**
tlbia [9, 10]	None	Context-synchronizing instruction or **sync**

[1] Synchronization requirements for changing the power conserving mode are implementation-dependent.

[2] The effect of altering the EE bit is immediate as follows:

- If an **mtmsr** clears EE, neither an external interrupt nor a decrementer interrupt can occur after the instruction is executed.
- If an **mtmsr** sets EE, when an external interrupt, decrementer interrupt, or higher priority interrupt exists, the corresponding interrupt occurs immediately after the **mtmsr** is executed, and before the next instruction is executed in the program that set MSR[EE].

[3] A context synchronizing instruction is required after modification of MSR[ME] to ensure that the modification takes effect for subsequent machine check interrupts, which may not be recoverable and therefore may not be context synchronizing.

[4] The alteration must not cause an implicit branch in physical address space. The physical address of the context-altering instruction and of each subsequent instruction, up to and including the next context synchronizing instruction, must be independent of whether the alteration has taken effect.

[5] Synchronization requirements for changing from one endian mode to the other are implementation-dependent.

[6] SDR1 must not be altered when MSR[DR] = 1 or MSR[IR] = 1; if it is, the results are undefined.

[7] A **sync** instruction is required before the **mtspr** instruction because SDR1 identifies the page table and thereby the location of the reference and change (R and C) bits. To ensure that R and C bits are updated in the correct page table, SDR1 must not be altered until all R and C bit updates due to instructions before the **mtspr** have completed. A **sync** instruction guarantees this synchronization of R and C bit updates, while neither a context synchronizing operation nor the instruction fetching mechanism does so.

[8] The elapsed time between the content of the decrementer becoming negative and the signaling of the decrementer interrupt is not defined.

[9] For data accesses, the context synchronizing instruction before the **tlbie**, or **tlbia** instruction ensures that all memory accesses, due to preceding instructions, have completed to a point at which they have reported all interrupts that may be caused. The context synchronizing instruction after the **tlbie**, or **tlbia** ensures that subsequent memory accesses do not use the TLB entries being invalidated. It does not ensure that all memory accesses previously translated by the TLB entries being invalidated have completed with respect to memory or, for **tlbie** or **tlbia**, that R and C bit updates associated with those memory accesses have completed; if these completions must be ensured, the **tlbie**, or **tlbia** must be followed by a **sync** instruction rather than by a context synchronizing instruction.

[10] Multiprocessor systems have other requirements to synchronize TLB invalidate.

Chapter 3
Operand Conventions

This chapter describes the operand conventions as they are represented in the user instruction set architecture (UISA) and virtual environment architecture (VEA). Detailed descriptions are provided of conventions used for storing values in registers and memory, accessing registers and representing data in these registers in both big- and little-endian modes. The floating-point data formats and exception conditions are also described. Refer to Appendix C, "Floating-Point Models," for more information on the implementation of the IEEE floating-point execution models.

3.1 Data Organization in Memory and Data Transfers

U

Bytes in memory are numbered consecutively starting with 0. Each number is the address of the corresponding byte. Memory operands may be bytes, half words, words, or double words, or, for the load and store multiple and the load and store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, its lowest-numbered byte). Operand length is implicit for each instruction.

3.1.1 Aligned and Misaligned Accesses

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. That is, the natural address of an operand is an integral multiple of its length. An operand not aligned at its natural boundary is considered misaligned. Instructions are always 4 bytes long and word-aligned. Table 3-1 shows operand characteristics for single-register memory access instructions.

Table 3-1. Memory Operand Alignment

Operand	Length	Aligned Addr(60–63) [1]
Byte	8 bits	xxxx
Half word	2 bytes	xxx0
Word	4 bytes	xx00
Double word	8 bytes	x000
Quad word (Although not permitted as operands, quad-word alignment is desirable for certain memory operands.)	16 bytes	0000

[1] An x in an address bit position indicates that the bit can be 0 or 1 independent of the state of other address bits.

The concept of alignment is also applied more generally to data in memory. For example, a 12-byte data item is said to be word-aligned if its address is a multiple of four.

Some instructions require their memory operands to have certain alignment. In addition, alignment may affect performance. For single-register memory access instructions, the best performance is obtained when memory operands are aligned.

3.1.2 Byte Ordering

If individual data items were indivisible, the concept of byte ordering would be unnecessary. The order of bits or groups of bits within the smallest addressable unit of memory is irrelevant, because nothing can be observed about such order. Order matters only when scalars, which the processor and programmer regard as indivisible quantities, can be made up of more than one addressable unit of memory.

The smallest addressable memory unit is the byte (8 bits), and scalars are composed of one or more sequential bytes. When a 32-bit scalar is moved from a register to memory, it occupies four consecutive bytes in memory, and a decision must be made regarding the order of these bytes in these four addresses.

Both big- and little-endian byte ordering are supported; the default is big-endian. Big- and little-endian byte orderings are described as follows:

- Big-endian byte ordering (default). For big-endian scalars, the most-significant byte (MSB) is stored at the lowest (or starting) address while the least-significant byte (LSB) is stored at the highest (or ending) address. This is called big-endian because the big end of the scalar comes first in memory.
- Little-endian byte ordering. For little-endian scalars, the LSB is stored at the lowest (or starting) address while the MSB is stored at the highest (or ending) address. This is called little-endian because the little end of the scalar comes first in memory.

3.1.3 Structure Mapping Examples

Figure 3-1 shows a C programming example that defines data structure *S* is used in this section to demonstrate how the bytes that comprise each element (*a*, *b*, *c*, *d*, *e*, and *f*) are mapped into memory. The structure contains scalars (shown in hexadecimal in the comments) and a sequence of characters, shown in single quote marks.

```
struct {
        int     a;      /* 0x1112_1314                  word            */
        double  b;      /* 0x2122_2324_2526_2728        double word     */
        char *  c;      /* 0x3132_3334                  word            */
        char    d[7];   /* 'L','M','N','O','P','Q','R'  array of bytes  */
        short   e;      /* 0x5152                       half word       */
        int     f;      /* 0x6162_6364                  word            */
} S;
```

Figure 3-1. C Program Example—Data Structure *S*

3.1.3.1 Big-Endian Mapping

Figure 3-2 shows the big-endian mapping of the structure. Note that the MSB of each scalar is at the lowest address. The mapping uses padding (indicated by (x)) to align the scalars—4 bytes between elements *a* and *b*, 1 byte between *d* and *e*, and 2 bytes between *e* and *f*. Note that the padding is determined by the compiler, not the architecture.

Contents	11	12	13	14	(x)	(x)	(x)	(x)
Address	00	01	02	03	04	05	06	07
Contents	21	22	23	24	25	26	27	28
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	31	32	33	34	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’	(x)	51	52	(x)	(x)
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	61	62	63	64	(x)	(x)	(x)	(x)
Address	20	21	22	23	24	25	26	27

Figure 3-2. Big-Endian Mapping of Structure *S*

3.1.3.2 Little-Endian Mapping

Figure 3-3 shows the structure using little-endian mapping. Note that the LSB of each scalar is at the lowest address.

Contents	14	13	12	11	(x)	(x)	(x)	(x)
Address	00	01	02	03	04	05	06	07
Contents	28	27	26	25	24	23	22	21
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	34	33	32	31	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’	(x)	52	51	(x)	(x)
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	64	63	62	61	(x)	(x)	(x)	(x)
Address	20	21	22	23	24	25	26	27

Figure 3-3. Little-Endian Mapping of Structure *S*

Figure 3-3 shows the sequence of double words laid out with addresses increasing from left to right. Programmers familiar with little-endian byte ordering may be more accustomed to viewing double words laid out with addresses increasing from right to left, as shown in Figure 3-4. This allows the little-endian programmer to view each scalar in its natural byte order of MSB to LSB. However, to demonstrate how

the PowerPC architecture provides both big- and little-endian support, this section uses the convention of showing addresses increasing from left to right, as in Figure 3-3.

Contents	(x)	(x)	(x)	(x)	11	12	13	14
Address	07	06	05	04	03	02	01	00
Contents	21	22	23	24	25	26	27	28
Address	0F	0E	0D	0C	0B	0A	09	08
Contents	‘O’	‘N’	‘M’	‘L’	31	32	33	34
Address	17	16	15	14	13	12	11	10
Contents	(x)	(x)	51	52	(x)	‘R’	‘Q’	‘P’
Address	1F	1E	1D	1C	1B	1A	19	18
Contents	(x)	(x)	(x)	(x)	61	62	63	64
Address	27	26	25	24	23	22	21	20

Figure 3-4. Little-Endian Mapping of Structure *S* —Alternate View

3.1.4 Byte Ordering in PowerPC Architecture

The architecture supports both big- and little-endian byte ordering; however, the code sequence to switch modes may differ among processors. Byte ordering is specified through two MSR bits. MSR[LE] (little-endian mode) indicates the endian mode in which the processor is currently operating; MSR[ILE] (exception little-endian mode) specifies the mode to be used when an exception handler is invoked. When an exception occurs, MSR[ILE] (as set for the interrupted process) is copied into MSR[LE] to select the endian mode for the context established by the exception. For both bits, a value of 0 specifies big-endian mode and a value of 1 specifies little-endian mode.

The architecture provides load and store instructions that reverse byte ordering. These instructions have the effect of loading and storing data in the endian mode opposite from that which the processor is operating. See Section 4.2.3.4, “Integer Load and Store with Byte-Reverse Instructions.”

3.1.4.1 Aligned Scalars in Little-Endian Mode

Chapter 4, “Addressing Modes and Instruction Set Summary,” describes the effective address calculation for the load and store instructions. For processors in little-endian mode, the effective address is modified before being used to access memory. The 3 low-order effective address bits are exclusive-ORed (XOR) with a 3-bit value that depends on operand length, as shown in Table 3-2. This modification is sometimes called munging.

Table 3-2. EA Modifications

Data Width (Bytes)	EA Modification
8	No change
4	XOR with 0b100
2	XOR with 0b110
1	XOR with 0b111

The modified physical address is passed to the cache or to main memory, and data of the specified width is transferred (in big-endian order—MSB at the lowest address and LSB at the highest) between a GPR or FPR and the addressed memory locations (as modified).

Modifying the address makes it appear to the processor that individual aligned scalars are stored as little-endian, when in fact they are stored in big-endian order, but at different byte addresses within double words. Only the address is modified, not the byte order.

Taking into account the preceding address modifications, in little-endian mode, structure *S* is placed in memory as shown in Figure 3-5.

Contents	(x)	(x)	(x)	(x)	11	12	13	14
Address	00	01	02	03	04	05	06	07
Contents	21	22	23	24	25	26	27	28
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	‘O’	‘N’	‘M’	‘L’	31	32	33	34
Address	10	11	12	13	14	15	16	17
Contents	(x)	(x)	51	52	(x)	‘R’	‘Q’	‘P’
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	(x)	(x)	(x)	(x)	61	62	63	64
Address	20	21	22	23	24	25	26	27

Figure 3-5. Modified Little-Endian Structure *S* as Seen by the Memory Subsystem

Note that the mapping shown in Figure 3-5 is not a true little-endian mapping of the structure *S*. However, because the processor modifies the address when accessing memory, the physical structure *S* shown in Figure 3-5 appears to the processor as the structure *S* shown in Figure 3-6.

Contents	14	13	12	11				
Address	00	01	02	03	04	05	06	07
Contents	28	27	26	25	24	23	22	21
Address	08	09	0A	0B	0C	0D	0E	0F
Contents	34	33	32	31	‘L’	‘M’	‘N’	‘O’
Address	10	11	12	13	14	15	16	17
Contents	‘P’	‘Q’	‘R’		52	51		
Address	18	19	1A	1B	1C	1D	1E	1F
Contents	64	63	62	61				
Address	20	21	22	23	24	25	26	27

Figure 3-6. Modified Little-Endian Structure *S* as Seen by the Processor

Note that the mapping in Figure 3-6 is identical to the little-endian mapping in Figure 3-3. However, from outside of the processor, the addresses of the bytes making up the structure *S* are as shown in Figure 3-5. These addresses match neither the big-endian mapping of Figure 3-2 nor the true little-endian mapping of Figure 3-3. This must be considered when performing I/O operations in little-endian mode, as described in Section 3.1.4.5, “Input/Output Data Transfer Addressing in Little-Endian Mode.”

3.1.4.2 Misaligned Scalars in Little-Endian Mode

Performing an XOR operation on the low-order bits of the address works only if the scalar is aligned on a boundary equal to a multiple of its length. Table 3-7 shows a true little-endian mapping of the four-byte word 0x1112_1314, stored at address 05.

Contents						14	13	12
Address	00	01	02	03	04	05	06	07
Contents	11							
Address	08	09	0A	0B	0C	0D	0E	0F

Figure 3-7. True Little-Endian Mapping, Word Stored at Address 05

For the true little-endian example in Figure 3-7, the LSB (0x14) is stored at address 0x05, the next byte (0x13) is stored at address 0x06, the third byte (0x12) is stored at address 0x07, and the MSB (0x11) is stored at address 0x08.

When a processor, in little-endian mode, issues a single-register load or store instruction with a misaligned effective address, it may take an alignment exception. In this case, a single-register load or store instruction means any of the integer load/store, load/store with byte-reverse, memory synchronization (excluding **sync**), or floating-point load/store (including **stfiwx**) instructions. Processors in little-endian mode are not

required to invoke an alignment exception when such a misaligned access is attempted. The processor may handle some or all such accesses without taking an alignment exception.

The architecture requires that half words, words, and double words be placed in memory such that the little-endian address of the lowest-order byte is the effective address computed by the load or store instruction; the little-endian address of the next-lowest-order byte is one greater, and so on. However, because processors in little-endian mode modify the effective address, the byte order of a misaligned scalar must be as if they were accessed one at a time.

Using the same example as shown in Figure 3-7, when the LSB (0x14) is stored to address 0x05, the address is XORed with 0b111 to become 0x02. When the next byte (0x13) is stored to address 0x06, the address is XORed with 0b111 to become 0x01. When the third byte (0x12) is stored to address 0x07, the address is XORed with 0b111 to become 0x00. Finally, when the MSB (0x11) is stored to address 0x08, the address is XORed with 0b111 to become 0x0F. Figure 3-8 shows the misaligned word, stored by a little-endian program, as seen by the memory subsystem.

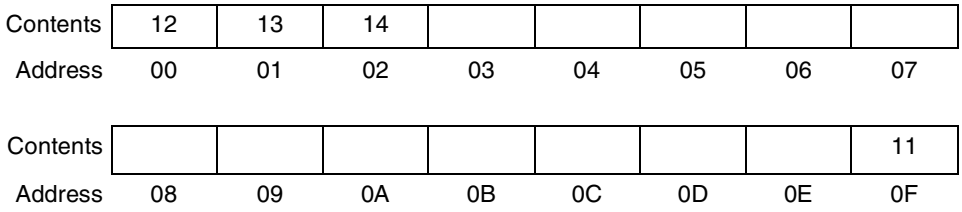


Figure 3-8. Word at Little-Endian Address 05 as Seen by the Memory Subsystem

Note that the misaligned word in this example spans two double words. The two parts of the misaligned word are not contiguous as seen by the memory system. An implementation may support some but not all misaligned little-endian accesses. For example, a misaligned little-endian access that is contained within a double word may be supported, while one that spans double words may cause an alignment exception.

3.1.4.3 Nonscalars

Two types of instructions handle nonscalars (multiple instances of scalars):

- Load and store multiple instructions
- Load and store string instructions

Address modification cannot be used because these instructions typically operate on more than one word-length scalar. These instructions cause alignment exception conditions when the processor executes in little-endian mode. String accesses are not supported, but they are inherently byte-based operations and can be broken into a series of word-aligned accesses.

3.1.4.4 Instruction Addressing in Little-Endian Mode

Instructions are word-aligned in memory. They are fetched as if the current instruction address is incremented by four for each sequential instruction. In little-endian mode, the instruction address is XORed with 0b100 as described in Section 3.1.4.1, “Aligned Scalars in Little-Endian Mode.” A program is thus an array of little-endian words with each word fetched and executed in order (not including branches).

Instruction addresses visible to an executing program are the effective addresses computed by that program, or, in the case of the exception handlers, effective addresses that were or could have been computed by the interrupted program. These addresses are independent of the endian mode. Examples for little-endian mode include the following:

- An instruction address placed in the link register by branch and link operation or an instruction address saved in an SPR when an exception is taken is the address that a program executing in little-endian mode would use to access the instruction as a word of data using a load instruction.
- An offset in a relative branch instruction reflects the difference between the addresses of the branch and target instructions, where the addresses used are those that a program executing in little-endian mode would use to access the instructions as data words using a load instruction.
- A target address in an absolute branch instruction is the address that a program executing in little-endian mode would use to access the target instruction as a word of data using a load instruction.
- Memory locations that contain the first set of instructions executed by each kind of exception handler must be set consistently with the endian mode in which the exception handler is invoked. Thus, if the handler is to be invoked in little-endian mode, the first set of instructions comprising each kind of exception handler must appear in memory with the instructions within each double word reversed from the order in which they are to be executed.

3.1.4.5 Input/Output Data Transfer Addressing in Little-Endian Mode

In big-endian mode, the processor and memory subsystem recognize the same byte as byte 0. However, this is not true in little-endian mode because of the address bits modified when the processor accesses memory.

I/O transfers in little-endian mode must be performed as if the bytes transferred were accessed one at a time, using the little-endian address modification appropriate for the single-byte transfers (that is, the lowest-order address bits must be XORed with 0b111). This does not mean that I/O operations in little-endian systems must be performed using only 1-byte transfers. Transfers can be as wide as desired, but the byte order within double words must be as if they were fetched or stored one at a time. That is, in a true little-endian I/O device, the system must provide a way to modify and unmodify addresses and reverse the bytes within a double word (MSB to LSB).

3.2 Operand Placement and Performance—VEA

The VEA states that the placement (location and alignment) of operands in memory affects the relative performance of memory accesses. The best performance is guaranteed if memory operands are aligned on natural boundaries. For more information on memory access ordering and atomicity, refer to Section 5.2, “The Virtual Environment.”

3.2.1 Summary of Performance Effects

For best performance across the widest range of PowerPC processor implementations, the programmer should assume the performance model described in Table 3-3 and Figure 3-4 with respect to the placement of memory operands.

The performance of accesses varies depending on the following:

- Operand size and alignment
- Endian mode (big-endian or little-endian)
- Whether a cache block, page, BAT, or segment boundary is crossed

Table 3-3 applies when the processor is in big-endian mode.

Table 3-3. Performance Effects of Memory Operand Placement, Big-Endian Mode

Operand			Boundary Crossing			
Type	**Size**	**Byte Alignment**	**None**	**Cache Block**	**Page**	**BAT/Segment**
Integer	8 byte	8 4 <4	Optimal Good Poor	— Good Poor	— Poor Poor	— Poor Poor
	4 byte	4 <4	Optimal Good	— Good	— Poor	— Poor
	2 byte	2 <2	Optimal Good	— Good	— Poor	— Poor
	1 byte	1	Optimal	—	—	—
	lmw, **stmw**	4	Good	Good	Good[1]	Poor
	String	—	Good	Good	Poor	Poor
Floating point	8 byte	8 4 <4	Optimal Good Poor	— Good Poor	— Poor Poor	— Poor Poor
	4 byte	4 <4	Optimal Poor	— Poor	— Poor	— Poor

1 Crossing a page boundary where the memory/cache access attributes of the two pages differ is equivalent to crossing a segment boundary and thus has poor performance.

Table 3-4 applies when the processor is in little-endian mode.

Table 3-4. Performance Effects of Memory Operand Placement, Little-Endian Mode

Operand			Boundary Crossing			
Type	**Size**	**Byte Alignment**	**None**	**Cache Block**	**Page**	**BAT/Segment**
Integer	8 byte	8 <8	Optimal Poor	— Poor	— Poor	— Poor
	4 byte	4 <4	Optimal Poor	— Poor	— Poor	— Poor
	2 byte	2 <2	Optimal Poor	— Poor	— Poor	— Poor
	1 byte	1	Optimal	—	—	—
Floating point	8 byte	8 <8	Optimal Poor	— Poor	— Poor	— Poor
	4 byte	4 <4	Optimal Poor	— Poor	— Poor	— Poor

The load/store multiple and the load/store string instructions are supported only in big-endian mode. Load/store multiple instructions are defined to operate only on aligned operands. Load/store string instructions have no alignment requirements.

3.2.2 Instruction Restart

The execution of a memory access instruction may abort after part of an access is performed for several reasons. For example, if a program attempts to access a page for the first time or when the processor must check for a change in the memory and cache access attributes when an access crosses a page boundary. When this occurs, the processor or operating system may restart the instruction, in which case, some bytes at that location may be loaded from or stored to the target location a second time.

The following rules apply to memory accesses with regard to restarting the instruction:

- Aligned accesses—A single-register instruction that accesses an aligned operand is never restarted (that is, it is not partially executed).
- Misaligned accesses—A single-register instruction that accesses a misaligned operand may be restarted if the access crosses a page, BAT, or segment boundary, or if the processor is in little-endian mode.
- Load/store multiple, load/store string instructions—These instructions may be restarted if, in accessing the locations specified by the instruction, a page, BAT, or segment boundary is crossed.

Programmers should assume that any misaligned access in a segment might be restarted. When the processor is in big-endian mode, software can ensure that misaligned accesses are not restarted by placing the misaligned data in BAT areas, as BAT areas have no internal protection boundaries. See Section 7.5, “Block Address Translation.”

3.3 Floating-Point Execution Models—UISA

U The architecture supports the two following types of floating-point instructions:

- Computational instructions including IEEE-754 defined operations for 64- and 32-bit arithmetic (addition, subtraction, multiplication, division, extracting the square root, rounding conversion, comparison, and combinations of these) and architecture-defined multiply-add and reciprocal estimate instructions.
- Noncomputational instructions—floating-point load, store, and move instructions.

Although computational and noncomputational instructions are governed by MSR[FP] (that allows floating-point instructions to be executed), only computational instructions are considered floating-point operations throughout this chapter.

The IEEE standard requires that single-precision arithmetic be provided for single-precision operands. The standard permits double-precision arithmetic instructions to have either (or both) single-precision or double-precision operands, but states that single-precision arithmetic instructions should not accept double-precision operands. The guidelines are as follows:

- Double-precision arithmetic instructions may have single-precision operands but always produce double-precision results.

- Single-precision arithmetic instructions require all operands to be single-precision and always produce single-precision results.

For arithmetic instructions, double- to single-precision conversion must be done explicitly by software; single- to double-precision conversion is done implicitly by the processor.

All implementations provide the equivalent of the following execution models to ensure that identical results are obtained. The definition of the arithmetic instructions for infinities, denormalized numbers, and NaNs follow conventions described in the following sections. Appendix C, “Floating-Point Models,” has additional detailed information on the execution models for IEEE operations as well as the other floating-point instructions.

Although the double-precision format specifies an 11-bit exponent, exponent arithmetic uses two additional bit positions to avoid potential transient overflow conditions. An extra bit is required when denormalized double-precision numbers are prenormalized. A second bit is required to permit computation of the adjusted exponent value in the following examples when the corresponding exception enable bit is 1 (exceptions are referred to as interrupts in the architecture specification):

- Underflow during multiplication using a denormalized operand
- Overflow during division using a denormalized divisor

3.3.1 Floating-Point Data Format

The UISA defines a 32-bit format for a single-precision floating-point value and a 64-bit format for a double-precision floating-point value. Floating-point data in memory may be in single- or double-precision format, however both single- and double-precision values in floating-point registers (FPRs) are stored in double-precision format.

Figure 3-9 shows single-precision format; Figure 3-10 shows double-precision format.

Figure 3-9. Floating-Point Single-Precision Format

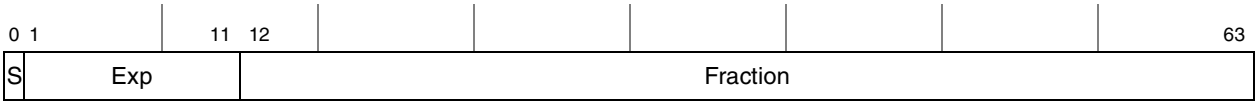


Figure 3-10. Floating-Point Double-Precision Format

Both formats consist of three fields:

- S (sign bit)
- EXP (exponent + bias)
- FRACTION (fraction)

If only a portion of a floating-point data item in memory is accessed, as with a load or store instruction for a byte or half word (or word in the case of floating-point double-precision format), the value affected depends on whether the system is using big- or little-endian byte ordering, as described in Section 3.1.2, “Byte Ordering.” Big-endian mode is the default.

For numeric values, the significand consists of a leading implied bit concatenated on the right with the FRACTION. This leading implied bit (the first bit to the left of the binary point) is 1 for normalized numbers and 0 for denormalized numbers. Values representable in the two floating-point formats can be specified by the parameters in Table 3-5.

Table 3-5. IEEE Floating-Point Fields

Parameter	Single-Precision	Double-Precision
Exponent bias	+127	+1023
Maximum exponent (unbiased)	+127	+1023
Minimum exponent (unbiased)	–126	–1022
Format width	32 bits	64 bits
Sign width	1 bit	1 bit
Exponent width	8 bits	11 bits
Fraction width	23 bits	52 bits
Significand width	24 bits	53 bits

As Table 3-6 shows, an exponent's true value can be determined by subtracting 127 for single-precision numbers and 1023 for double-precision numbers. Note that an exponent of all ones indicates an infinity or NaN; all zeros indicates zero or a denormalized number.

Table 3-6. Biased Exponent Format

Biased Exponent (Binary)	Single-Precision (Unbiased)	Double-Precision (Unbiased)
11. 11	Reserved for infinities and NaNs	
11. 10	+127	+1023
11. 01	+126	+1022
⋮	⋮	⋮
10. 00	1	1
01. 11	0	0
01. 10	–1	–1
⋮	⋮	⋮
00. 01	–126	–1022
00. 00	Reserved for zeros and denormalized numbers	

3.3.1.1 Value Representation

The UISA defines numerical and nonnumerical values representable in single- and double-precision formats. Numerical values are approximations to real numbers, including normalized numbers, denormalized numbers, and zero values. Representable nonnumerical values are positive and negative

infinities and NaNs. Infinities are adjoined to the real numbers but are not numbers themselves, and the standard rules of arithmetic do not hold when they appear in an operation. They are related to the real numbers by order alone. It is possible, however, to define restricted operations among numbers and infinities as defined below. Figure 3-11 shows the relative location of defined numerical entities on a real number line. Tiny values include denormalized numbers and numbers that too small to be represented for a particular precision format, but do not include ±0.

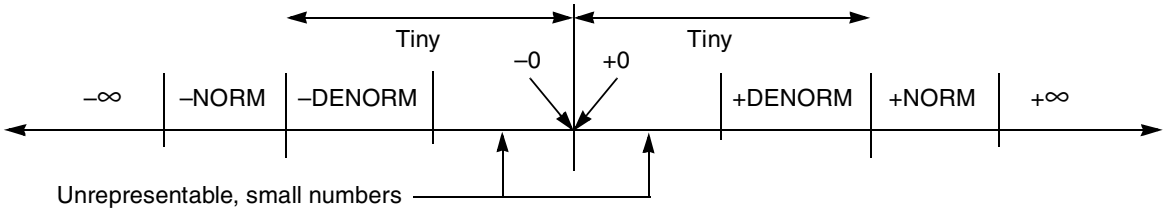


Figure 3-11. Approximation to Real Numbers

Positive and negative NaNs convey diagnostic information such as representation of uninitialized variables and are not related to the numbers, ±∞, or each other by order or value. Table 3-7 describes each of the floating-point formats.

Table 3-7. Recognized Floating-Point Numbers

Sign Bit	Biased Exponent	Implied Bit	Fraction	Value
0	Maximum	x	Nonzero	NaN
0	Maximum	x	Zero	+Infinity
0	0 < Exponent < Maximum	1	x	+Normalized
0	0	0	Nonzero	+Denormalized
0	0	x	Zero	+0
1	0	x	Zero	–0
1	0	0	Nonzero	–Denormalized
1	0 < Exponent < Maximum	1	x	–Normalized
1	Maximum	x	Zero	–Infinity
1	Maximum	x	Nonzero	NaN

The following sections describe floating-point values defined in the architecture.

3.3.1.2 Binary Floating-Point Numbers

Binary floating-point numbers are machine-representable approximations of real numbers. Three categories are supported—normalized numbers, denormalized numbers, and zeros.

3.3.1.3 Normalized Numbers (±NORM)

The values for normalized numbers have a biased exponent value in the range:

- 1–254 in single-precision format
- 1–2046 in double-precision format

The implied unit bit is one. Normalized numbers are interpreted as follows:

$$NORM = (-1)^{s} \times 2^{E} \times (1.fraction)$$

The variable (s) is the sign, (E) is the unbiased exponent, and (1.fraction) is the significand composed of a leading unit bit (implied bit) and a fractional part. Figure 3-12 shows the format for normalized numbers.

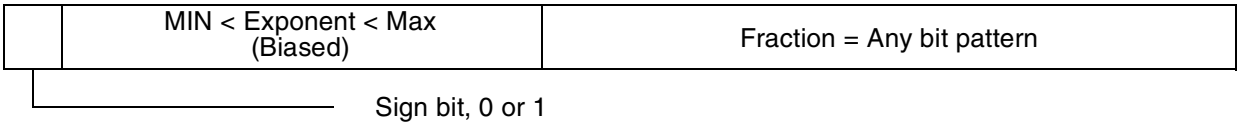


Figure 3-12. Format for Normalized Numbers

The ranges covered by the magnitude (M) of a normalized floating-point number are approximated in the following decimal representation:

Single-precision format:

$$1.2 \times 10^{-38} \leq M \leq 3.4 \times 10^{38}$$

Double-precision format:

$$2.2 \times 10^{-308} \leq M \leq 1.8 \times 10^{308}$$

3.3.1.4 Zero Values (±0)

Zero values, Figure 3-13, have a biased exponent value of zero and fraction of zero. Zeros can have a positive or negative sign. Comparison operations ignore the sign (that is, +0 = –0). Arithmetic with zero results is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 3.3.6.1.1, “Invalid Operation Exception Condition.” Rounding a zero result affects only the sign (±0).

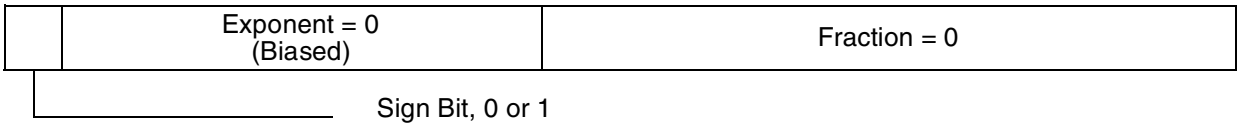


Figure 3-13. Format for Zero Numbers

3.3.1.5 Denormalized Numbers (±DENORM)

Denormalized numbers have a biased exponent value of zero and a nonzero fraction. The format for denormalized numbers is shown in Figure 3-14.

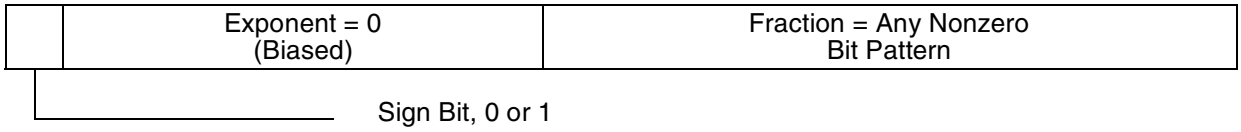


Figure 3-14. Format for Denormalized Numbers

Denormalized numbers are nonzero numbers smaller in magnitude than the normalized numbers. They are values in which the implied unit bit is zero. Denormalized numbers are interpreted as follows:

$$DENORM = (-1)^{s} \times 2^{Emin} \times (0.fraction)$$

The value Emin is the minimum unbiased exponent value for a normalized number (–126 for single-precision, –1022 for double-precision).

3.3.1.6 Infinities (±∞)

Infinities have a maximum biased exponent value of 255 in single-precision format, 2047 in double-precision format, and a zero fraction value. Infinities approximate values greater in magnitude than the maximum normalized value. Infinity arithmetic is defined as the limiting case of real arithmetic, with restricted operations defined among numbers and infinities. Infinities and real numbers can be related by ordering in the affine sense:

–∞ < every finite number < +∞

The format for infinities is shown in Figure 3-15.

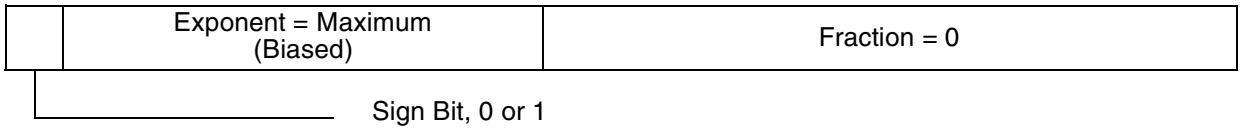


Figure 3-15. Format for Positive and Negative Infinities

Arithmetic using infinite numbers is always exact and does not signal any exception, except when an exception occurs due to the invalid operations as described in Section 3.3.6.1.1, “Invalid Operation Exception Condition.”

3.3.1.7 Not a Numbers (NaNs)

NaNs have the maximum biased exponent value and a nonzero fraction. The format for NaNs is shown in Figure 3-16. The sign bit of NaN does not show an algebraic sign; rather, it is simply another bit in the NaN. If the highest-order bit of the fraction field is a zero, the NaN is a signaling NaN; otherwise it is a quiet NaN (QNaN).

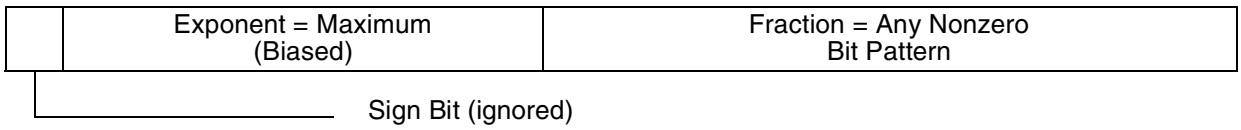


Figure 3-16. Format for NaNs

Signaling NaNs signal exceptions when they are specified as arithmetic operands.

QNaNs represent the results of certain invalid operations, such as attempts to perform arithmetic operations on infinities or NaNs, when the invalid operation exception is disabled (FPSCR[VE] = 0). QNaNs propagate through all operations, except floating-point round to single-precision, ordered comparison, and conversion to integer operations. They signal exceptions only for ordered comparison and conversion to integer operations. Specific encodings in QNaNs can thus be preserved through a sequence of operations and used to convey diagnostic information to help identify results from invalid operations.

When a QNaN results from an operation because an operand is a NaN or because a QNaN is generated due to a disabled invalid operation exception, the following rule is applied to determine the QNaN to be stored as the result:

```
If (frA) is a NaN
  Then frD ← (frA)
  Else if (frB) is a NaN
    Then if instruction is frsp
      Then frD ← (frB)[0:34]||(29)0
      Else frD ← (frB)
    Else if (frC) is a NaN
      Then frD ← (frC)
      Else if generated QNaN
        Then frD ← generated QNaN
```

If the operand specified by **fr**A is a NaN, that NaN is stored as the result. Otherwise, if the operand specified by **fr**B is a NaN (if the instruction specifies an **fr**B operand), that NaN is stored as the result, with the low-order 29 bits cleared. Otherwise, if the operand specified by **fr**C is a NaN (if the instruction specifies an **fr**C operand), that NaN is stored as the result. Otherwise, if a QNaN is generated by a disabled invalid operation exception, that QNaN is stored as the result. If a QNaN is to be generated as a result, the QNaN generated has a sign bit of zero, an exponent field of all ones, and a highest-order fraction bit of one with all other fraction bits zero. An instruction that generates a QNaN as the result of a disabled invalid operation generates this QNaN. This is shown in Figure 3-17.

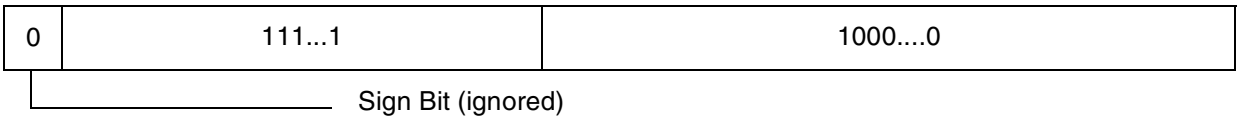


Figure 3-17. Representation of Generated QNaN

3.3.2 Sign of Result

The following rules govern the sign of the result of an arithmetic operation, when the operation does not yield an exception. These rules apply even when the operands or results are ±0 or ±∞:

- The sign of the result of an addition operation is the sign of the source operand having the larger absolute value. If both operands have the same sign, the sign of the result of an addition operation is the same as the sign of the operands. The sign of the result of the subtraction operation, x – y, is the same as the sign of the result of the addition operation, x + (–y).
- When the sum of two operands with opposite sign, or the difference of two operands with the same sign, is exactly zero, the sign of the result is positive in all rounding modes except round toward negative infinity (–∞), in which case the sign is negative.

- The sign of the result of a multiplication or division operation is the XOR of the signs of the source operands.
- The sign of the result of a round to single-precision or convert to/from integer operation is the sign of the source operand.
- The sign of the result of a square root or reciprocal square root estimate operation is always positive, except that the square root of –0 is –0 and the reciprocal square root of –0 is –infinity.

For multiply-add instructions, these rules are applied first to the multiplication operation and then to the addition or subtraction operation (one of the source operands to the addition or subtraction operation is the result of the multiplication operation).

3.3.3 Normalization and Denormalization

The intermediate result of an arithmetic or Floating Round to Single-Precision (**frsp***x*) instruction may require normalization and/or denormalization. When an intermediate result consists of a sign bit, an exponent, and a nonzero significand with a zero leading bit, the result must be normalized (and rounded) before being stored to the target.

A number is normalized by shifting its significand left and decrementing its exponent by one for each bit shifted until the leading significand bit becomes one. The guard and round bits are also shifted, with zeros shifted into the round bit; see Section C.1, “Execution Model for IEEE Operations,” for information about the guard and round bits. During normalization, the exponent is regarded as if its range were unlimited.

If an intermediate result has a nonzero significand and an exponent that is smaller than the minimum value that can be represented in the format specified for the result, this value is referred to as ‘tiny’ and the stored result is determined by the rules described in Section 3.3.6.2.2, “Underflow Exception Condition.” These rules may involve denormalization. The sign of the number does not change.

An exponent can become tiny in either of the following circumstances:

- As the result of an arithmetic or **frsp***x* instruction
- As the result of decrementing the exponent in the process of normalization.

Normalization is the process of coercing the leading significand bit to be a 1 while denormalization is the process of coercing the exponent into the target format's range. In denormalization, the significand is shifted to the right while the exponent is incremented for each bit shifted until the exponent equals the format’s minimum value. The result is then rounded. If any significand bits are lost due to the rounding of the shifted value, the result is considered inexact. The sign of the number does not change.

3.3.4 Data Handling and Precision

There are specific instructions for moving floating-point data between the FPRs and memory. For double-precision format data, the data is not altered during the move. For single-precision data, the format is converted to double-precision format when data is loaded from memory into an FPR. A format conversion from double- to single-precision is performed when data from an FPR is stored as single-precision. These operations do not cause floating-point exceptions.

All floating-point arithmetic, move, and select instructions use floating-point double-precision format. Floating-point single-precision formats are obtained by using the following four types of instructions:

- Load floating-point single-precision instructions—These instructions access a single-precision operand in single-precision format in memory, convert it to double-precision, and load it into an FPR. Floating-point exceptions do not occur during the load operation.
- Floating Round to Single-Precision (**frsp***x*) instruction—The **frsp***x* instruction rounds a double-precision operand to single-precision, checking the exponent for single-precision range and handling any exceptions according to respective FPSCR enable bits. The instruction places that operand into an FPR as a double-precision operand. For results produced by single-precision arithmetic instructions and by single-precision loads, this operation does not alter the value.
- Single-precision arithmetic instructions—These instructions take operands from the FPRs in double-precision format, perform the operation as if it produced an intermediate result correct to infinite precision and with unbounded range, and then force this intermediate result to fit in single-precision format. Status bits in the FPSCR and CR are set to reflect the single-precision result. The result is then converted to double-precision format and placed into an FPR. The result falls within the range supported by the single-precision format.

  Source operands for these instructions must be representable in single-precision format. Otherwise, the result placed into the target FPR and the setting of status bits, FPSCR and CR if update mode is selected, are undefined.
- Store floating-point single-precision instructions—These instructions convert a double-precision operand to single-precision format and store that operand into memory. If the operand requires denormalization in order to fit in single-precision format, it is automatically denormalized prior to being stored. No exceptions are detected on the store operation (the value being stored is effectively assumed to be the result of an instruction of one of the preceding three types).

When the result of a Load Floating-Point Single (**lfs**), Floating Round to Single-Precision (**frsp***x*), or single-precision arithmetic instruction is stored in an FPR, the low-order 29 fraction bits are zero. This is shown in Figure 3-18.

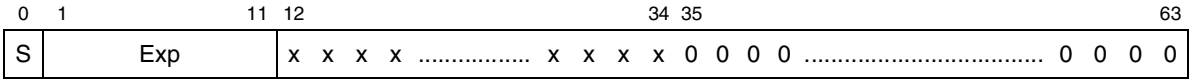


Figure 3-18. Single-Precision Representation in an FPR

The **frsp***x* instruction allows conversion from double- to single-precision with appropriate exception checking and rounding. It is used to convert double-precision floating-point values (produced by double-precision load and arithmetic instructions) to single-precision values before storing them into single-format memory elements or using them as operands for single-precision arithmetic instructions. Values produced by single-precision load and arithmetic instructions can be stored directly, or used directly as operands for single-precision arithmetic instructions, without being preceded by an **frsp***x* instruction.

A single-precision value can be used in double-precision arithmetic operations. The reverse is true only if the double-precision value can be represented in single-precision format. If double-precision accuracy is not required, using single-precision data and instructions may speed operations in some implementations.

3.3.5 Rounding

All arithmetic, rounding, and conversion instructions defined by the architecture (except the optional Floating Reciprocal Estimate Single (**fres***x*) and Floating Reciprocal Square Root Estimate (**frsqrte***x*) instructions) produce an intermediate result considered to be infinitely precise and with unbounded exponent range. This intermediate result is normalized or denormalized if required, and then rounded to the destination format. The final result is then placed into the target FPR in the double-precision format or in fixed-point format, depending on the instruction.

The IEEE-754 specification allows loss of accuracy to be defined as when the rounded result differs from the infinitely precise value with unbounded range (same as the definition of inexact). In the PowerPC architecture, this is how loss of accuracy is detected.

Let Z be the intermediate result (with infinite precision and unbounded range) or the operand of a conversion operation. If Z can be represented exactly in the target format, the result in all rounding modes is exactly Z. If Z cannot be represented exactly in the target format, let Z1 and Z2 be the next larger and next smaller numbers representable in the target format that bound Z; Z1 or Z2 can be used to approximate the result in the target format.

Figure 3-19 shows a graphical representation of Z, Z1, and Z2 in this case.

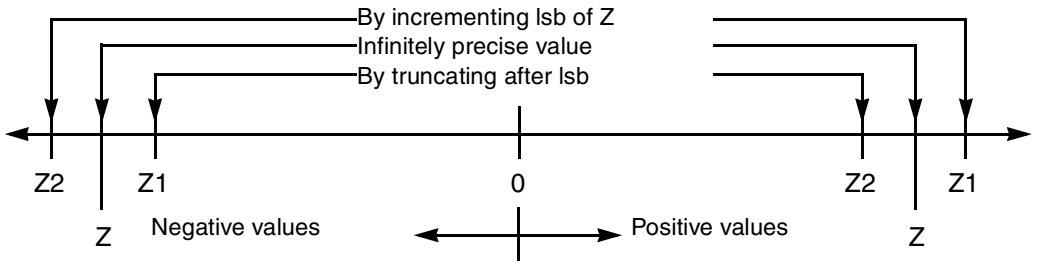


Figure 3-19. Relation of Z1 and Z2

Table 3-8 describes the four rounding modes available through FPSCR[RN].

Table 3-8. FPSCR[RN] Setting

RN	Rounding Mode	Rules
00	Round to nearest	Choose the best approximation (Z1 or Z2). If a tie, choose the one that is even (lsb 0).
01	Round toward zero	Choose the smaller in magnitude (Z1 or Z2).
10	Round toward +∞	Choose Z1.
11	Round toward –∞	Choose Z2.

See Section C.1, “Execution Model for IEEE Operations,” for a detailed explanation of rounding. Rounding occurs before an overflow condition is detected. This means that while an infinitely precise value with unbounded exponent range may be greater than the greatest representable value, the rounding mode may allow that value to be rounded to a representable value. In this case, no overflow condition occurs.

However, the underflow condition is tested before rounding. Therefore, if the value that is infinitely precise and with unbounded exponent range falls within the range of unrepresentable values, the

underflow condition occurs. The results in these cases are defined in Section 3.3.6.2.2, “Underflow Exception Condition.” Figure 3-20 shows the selection of Z1 and Z2 for the four possible rounding modes provided by FPSCR[RN].

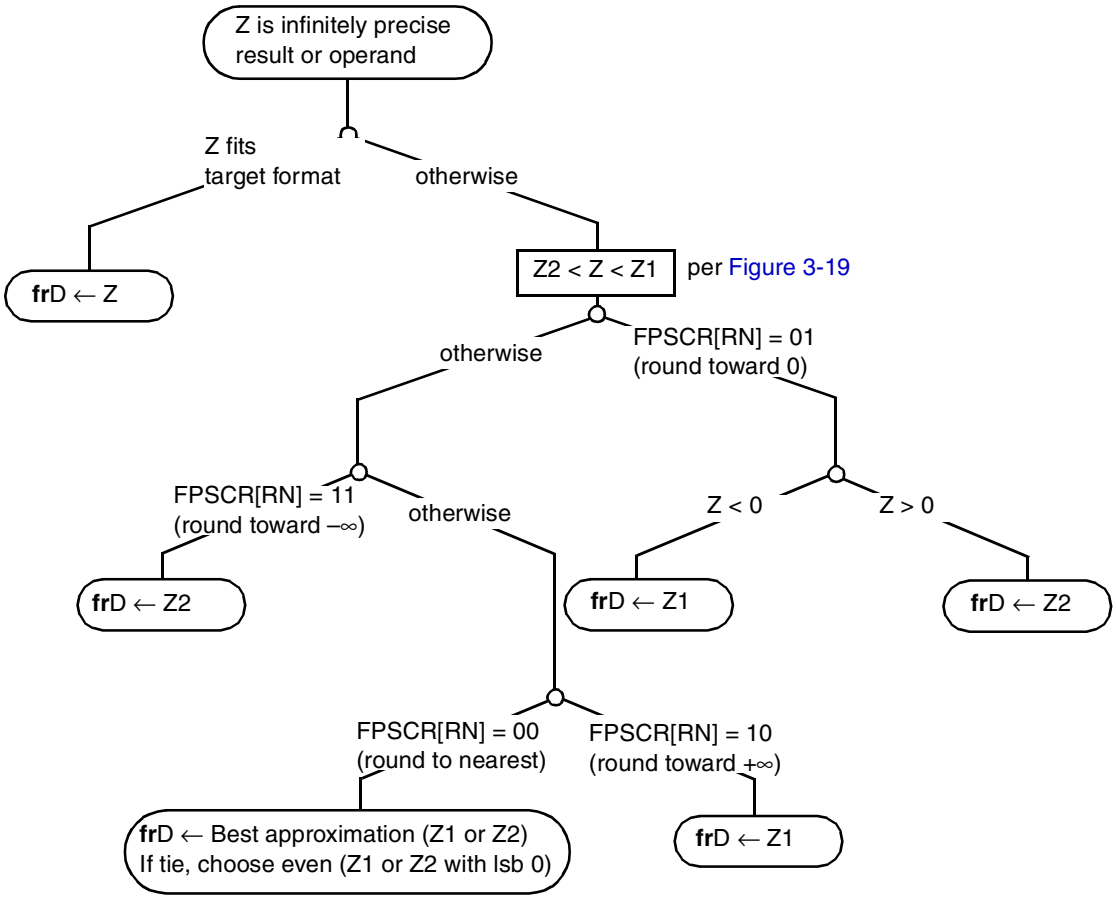


Figure 3-20. Selection of Z1 and Z2 for the Four Rounding Modes

All arithmetic, rounding, and conversion instructions affect FPSCR bits FR and FI, according to whether the rounded result is inexact (FI) and whether the fraction was incremented (FR) as shown in Figure 3-21. If the rounded result is inexact, FI is set and FR may be either set or cleared. If rounding does not change the result, both FR and FI are cleared. The optional **fres***x* and **frsqrte***x* instructions set FI and FR to undefined values; other floating-point instructions do not alter FR and FI.

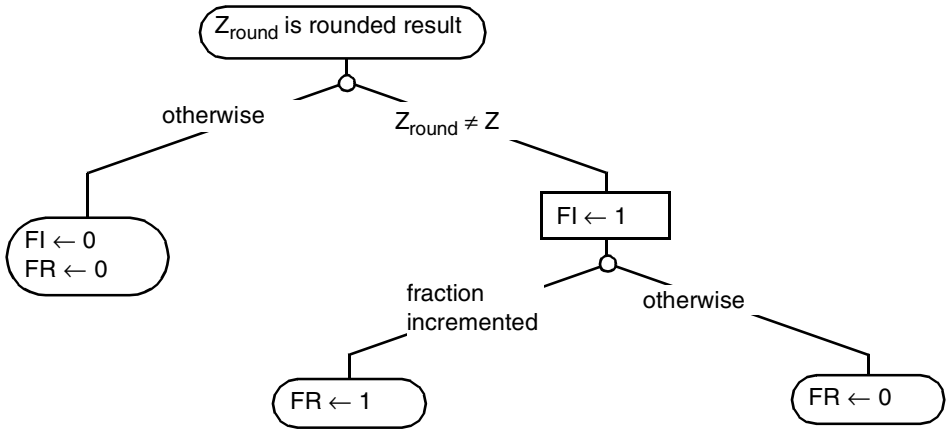


Figure 3-21. Rounding Flags in FPSCR

3.3.6 Floating-Point Program Exceptions

Only computational instructions can cause floating-point enabled exceptions (subsets of the program exception). Floating-point program exceptions are signaled by FPSCR condition bits described here and in Chapter 2, “Register Set.” These bits correspond to conditions identified as IEEE floating-point exceptions and can cause the system floating-point enabled exception error handler to be invoked. Handling for floating-point exceptions is described in Section 6.5.7, “Program Interrupt (0x00700).”

The FPSCR is shown in Figure 3-22.

Access: User Read/Write

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15–19	20	21	22	23	24	25	26	27	28	29	30	31
R W	FX	FEX	VX	OX	UX	ZX	XX	VXSNAN	VXISI	VXIDI	VXZDZ	VXIMZ	VXVC	FR	FI	FPRF	—	VXSOFT	VXSQRT	VXCVI	VE	OE	UE	ZE	XE	NI	RN	
Reset	All zeros																											

Figure 3-22. Floating-Point Status and Control Register (FPSCR)

Table 3-9 describes FPSCR bit settings.

Table 3-9. FPSCR Bit Settings

Bits	Name	Description
0	FX	Floating-point exception summary. Every floating-point instruction, except **mtfsfi** and **mtfsf**, implicitly sets FPSCR[FX] if that instruction causes any of the floating-point exception bits in the FPSCR to transition from 0 to 1. The **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, and **mtfsb1** instructions can alter FPSCR[FX] explicitly. This is a sticky bit.
1	FEX	Floating-point enabled exception summary. This bit signals the occurrence of any of the enabled exception conditions. It is the logical OR of all the floating-point exception bits masked by their respective enable bits (FEX = (VX & VE) ^ (OX & OE) ^ (UX & UE) ^ (ZX & ZE) ^ (XX & XE)). The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[FEX] explicitly. This is not a sticky bit.

Table 3-9. FPSCR Bit Settings (continued)

Bits	Name	Description
2	VX	Floating-point invalid operation exception summary. This bit signals the occurrence of any invalid operation exception. It is the logical OR of all of the invalid operation exception bits as described in Section 3.3.6.1.1, "Invalid Operation Exception Condition." The **mcrfs**, **mtfsf**, **mtfsfi**, **mtfsb0**, and **mtfsb1** instructions cannot alter FPSCR[VX] explicitly. This is not a sticky bit.
3	OX	Floating-point overflow exception. This is a sticky bit. See Section 3.3.6.2, "Overflow, Underflow, and Inexact Exception Conditions."
4	UX	Floating-point underflow exception. This is a sticky bit. See Section 3.3.6.2.2, "Underflow Exception Condition."
5	ZX	Floating-point zero divide exception. This is a sticky bit. See Section 3.3.6.1.2, "Zero Divide Exception Condition."
6	XX	Floating-point inexact exception. This is a sticky bit. See Section 3.3.6.2.3, "Inexact Exception Condition." XX is the sticky version of FPSCR[FI]. The following describes how XX is set by a given instruction: • If the instruction affects FPSCR[FI], the new value of FPSCR[XX] is obtained by logically ORing the old value of FPSCR[XX] with the new value of FPSCR[FI]. • If the instruction does not affect FPSCR[FI], the value of FPSCR[XX] is unchanged.
7	VXSNAN	Floating-point invalid operation exception for SNaN. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
8	VXISI	Floating-point invalid operation exception for ∞ – ∞. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
9	VXIDI	Floating-point invalid operation exception for ∞ ÷ ∞. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
10	VXZDZ	Floating-point invalid operation exception for 0 ÷ 0. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
11	VXIMZ	Floating-point invalid operation exception for ∞ * 0. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
12	VXVC	Floating-point invalid operation exception for invalid compare. This is a sticky bit. See Section 3.3.6.1.1, "Invalid Operation Exception Condition."
13	FR	Floating-point fraction rounded. The last arithmetic, rounding, or conversion instruction incremented the fraction. See Section 3.3.5, "Rounding." This bit is not sticky.
14	FI	Floating-point fraction inexact. The last arithmetic, rounding, or conversion instruction either produced an inexact result during rounding or caused a disabled overflow exception. See Section 3.3.5, "Rounding." This is not a sticky bit. For more information regarding the relationship between FPSCR[FI] and FPSCR[XX], see the description of the FPSCR[XX] bit.

Table 3-9. FPSCR Bit Settings (continued)

Bits	Name	Description
15–19	FPRF	Floating-point result flags. For arithmetic, rounding, and conversion instructions, FPRF is based on the result placed into the target register, except that if any portion of the result is undefined, the value placed here is undefined. 15 Floating-point result class descriptor (C). Arithmetic, rounding, and conversion instructions may set this bit with the FPCC bits to indicate the class of the result as shown in Table 3-10. Bits 16–19 comprise the floating-point condition code (FPCC). Floating-point compare instructions always set one of the FPCC bits to one and the other three FPCC bits to zero. Arithmetic, rounding, and conversion instructions may set the FPCC bits with the C bit to indicate the class of the result. Note that in this case the high-order three bits of the FPCC retain their relational significance indicating that the value is less than, greater than, or equal to zero. 16 Floating-point less than or negative (FL or <) 17 Floating-point greater than or positive (FG or >) 18 Floating-point equal or zero (FE or =) 19 Floating-point unordered or NaN (FU or ?) Note that these are not sticky bits.
20	—	Reserved
21	VXSOFT	Floating-point invalid operation exception for software request. This is a sticky bit. This bit can be altered only by the **mcrfs**, **mtfsfi**, **mtfsf**, **mtfsb0**, or **mtfsb1** instructions. For more detailed information, refer to Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
22	VXSQRT	Floating-point invalid operation exception for invalid square root. This is a sticky bit. For more detailed information, refer to Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
23	VXCVI	Floating-point invalid operation exception for invalid integer convert. This is a sticky bit. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
24	VE	Floating-point invalid operation exception enable. See Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
25	OE	IEEE floating-point overflow exception enable. See Section 3.3.6.2, “Overflow, Underflow, and Inexact Exception Conditions.”
26	UE	IEEE floating-point underflow exception enable. See Section 3.3.6.2.2, “Underflow Exception Condition.”
27	ZE	IEEE floating-point zero divide exception enable. See Section 3.3.6.1.2, “Zero Divide Exception Condition.”
28	XE	Floating-point inexact exception enable. See Section 3.3.6.2.3, “Inexact Exception Condition.”
29	NI	Floating-point non-IEEE mode. If this bit is set, results need not conform with IEEE standards and the other FPSCR bits may have meanings other than those described here. If the bit is set and if all implementation-specific requirements are met and if an IEEE-conforming result of a floating-point operation would be a denormalized number, the result produced is zero (retaining the sign of the denormalized number). Any other effects associated with setting this bit are described in the user’s manual for the implementation. Effects of the setting of this bit are implementation-dependent.
30–31	RN	Floating-point rounding control. See Section 3.3.5, “Rounding.” 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

Table 3-10 describes the floating-point result flags, FPSCR[FPRF].

Table 3-10. Floating-Point Result Flags—FPSCR[FPRF]

Result Flags (Bits 15–19)					Result Value Class
C	<	>	=	?	
1	0	0	0	1	Quiet NaN
0	1	0	0	1	–Infinity
0	1	0	0	0	–Normalized number
1	1	0	0	0	–Denormalized number
1	0	0	1	0	–Zero
0	0	0	1	0	+Zero
1	0	1	0	0	+Denormalized number
0	0	1	0	0	+Normalized number
0	0	1	0	1	+Infinity

The following conditions that can cause program exceptions are detected by the processor. These conditions may occur during execution of computational floating-point instructions. The corresponding bits set in the FPSCR are indicated in parentheses:

- Invalid operation exception condition (VX)
  - SNaN condition (VXSNAN)
  - Infinity – infinity condition (VXISI)
  - Infinity ÷ infinity condition (VXIDI)
  - Zero ÷ zero condition (VXZDZ)
  - Infinity * zero condition (VXIMZ)
  - Invalid compare condition (VXVC)
  - Software request condition (VXSOFT)
  - Invalid integer convert condition (VXCVI)
  - Invalid square root condition (VXSQRT)

  These exception conditions are described in Section 3.3.6.1.1, “Invalid Operation Exception Condition.”
- Zero divide exception condition (ZX). These exception conditions are described in Section 3.3.6.1.2, “Zero Divide Exception Condition.”
- Overflow Exception Condition (OX). These exception conditions are described in Section 3.3.6.2.1, “Overflow Exception Condition.”
- Underflow Exception Condition (UX). These exception conditions are described in Section 3.3.6.2.2, “Underflow Exception Condition.”
- Inexact Exception Condition (XX). These exception conditions are described in Section 3.3.6.2.3, “Inexact Exception Condition.”

Each floating-point exception condition and each category of invalid IEEE floating-point operation exception condition has a corresponding exception bit in the FPSCR. Generally, the occurrence of an exception condition depends only on the instruction and its arguments (with one deviation, described below). When one or more exception conditions arise during the execution of an instruction, the way in which the instruction completes execution depends on the corresponding IEEE floating-point enable bits in the FPSCR. If no governing enable bit is set, the instruction delivers a default result. Otherwise, specific condition bits and FPSCR[FX] are set and instruction execution is completed by suppressing or delivering a result. Finally, after instruction execution completes, a nonzero FPSCR[FX] causes a program exception if either MSR[FE0] or MSR[FE1] is set (invoking the system error handler). The FPR values immediately after the occurrence of an enabled exception do not depend on MSR[FE0,FE1].

FPSCR[FX] is set by any floating-point instruction (except **mtfsfi** and **mtfsf**) that causes any FPSCR exception bit to change from 0 to 1, or by **mtfsfi**, **mtfsf**, and **mtfsb1** instructions that explicitly set one of these bits. FPSCR[FEX] is set when an exception condition bits is set and the exception enable bit is one.

A single instruction can set multiple exception condition bits only in the following cases:

- The inexact exception condition bit (FPSCR[XX]) may be set with the overflow exception condition bit (FPSCR[OX]).
- The inexact exception condition bit (FPSCR[XX]) may be set with the underflow exception condition bit (FPSCR[UX]).
- The invalid IEEE floating-point operation exception condition bit (SNaN) may be set with invalid IEEE floating-point operation exception condition bit (∞*0) (FPSCR[VXIMZ]) for multiply-add instructions.
- The invalid operation exception condition bit (SNaN) may be set with the invalid IEEE floating-point operation exception condition bit (invalid compare) (FPRSC[VXVC]) for compare ordered instructions.
- The invalid IEEE floating-point operation exception condition bit (SNaN) may be set with the invalid IEEE floating-point operation exception condition bit (invalid integer convert) (FPSCR[VXCVI]) for convert-to-integer instructions.

Instruction execution is suppressed for the following kinds of exception conditions, so that there is no possibility that one of the operands is lost:

- Enabled invalid IEEE floating-point operation
- Enabled zero divide

For the remaining kinds of exception conditions, a result is generated and written to the destination specified by the instruction causing the exception condition. The result may depend on whether the condition is enabled or disabled. The kinds of exception conditions that deliver a result are the following:

- Disabled invalid IEEE floating-point operation
- Disabled zero divide
- Disabled overflow
- Disabled underflow
- Disabled inexact
- Enabled overflow

- Enabled underflow
- Enabled inexact

Subsequent sections define each of the floating-point exception conditions and specify the action taken when they are detected.

The IEEE standard specifies the handling of exception conditions in terms of traps and trap handlers. An FPSCR exception enable bit being set causes generation of the result value specified in the IEEE standard for the trap enabled case—the expectation is that the exception is detected by software, which revises the result. An FPSCR exception enable bit of 0 causes generation of the default result value specified for the trap disabled (or no trap occurs or trap is not implemented) case—the expectation is that the exception is not detected by software, which uses the default result. The result to be delivered in each case for each exception is described in the following sections.

The IEEE default behavior when an exception occurs, which is to generate a default value and not to notify software, is obtained by clearing all FPSCR exception enable bits and using ignore exceptions mode (see Table 3-11). In this case the system floating-point enabled exception error handler is not invoked, even if floating-point exceptions occur. If necessary, software can inspect the FPSCR exception bits to determine whether exceptions have occurred.

If the system error handler is to be invoked, the corresponding FPSCR exception enable bit must be set and a mode other than ignore exceptions mode must be used. In this case the system floating-point enabled exception error handler is invoked if an enabled floating-point exception condition occurs.

Whether and how the system floating-point enabled exception error handler is invoked if an enabled floating-point exception occurs is controlled by MSR bits FE0 and FE1 as shown in Table 3-11. (The system floating-point enabled exception error handler is never invoked if the appropriate floating-point exception is disabled.)

Table 3-11. MSR[FE0] and MSR[FE1] Bit Settings for FP Exceptions

FE0	FE1	Description
0	0	Ignore exceptions mode—Floating-point exceptions do not cause the program exception error handler to be invoked.
0	1	Imprecise nonrecoverable mode—When an exception occurs, the exception handler is invoked at some point at or beyond the instruction that caused the exception. It may not be possible to identify the excepting instruction or the data that caused the exception. Results from the excepting instruction may have been used by or affected subsequent instructions executed before the exception handler was invoked.
1	0	Imprecise recoverable mode—When an enabled exception occurs, the floating-point enabled exception handler is invoked at some point at or beyond the instruction that caused the exception. Sufficient information is provided to the exception handler that it can identify the excepting instruction and correct any faulty results. In this mode, no results caused by the excepting instruction have been used by or affected subsequent instructions that are executed before the exception handler is invoked.
1	1	Precise mode—The system floating-point enabled exception error handler is invoked precisely at the instruction that caused the enabled exception.

In precise mode, whenever the system floating-point enabled exception error handler is invoked, the architecture ensures that all instructions logically residing before the excepting instruction have completed and no instruction after the excepting instruction has been executed. In an imprecise mode, the instruction

flow may not be interrupted at the point of the instruction that caused the exception. The instruction at which the system floating-point exception handler is invoked has not been executed unless it is the excepting instruction and the exception is not suppressed.

In either of the imprecise modes, an FPSCR instruction can be used to force the occurrence of any invocations of the floating-point enabled exception handler, due to instructions initiated before the FPSCR instruction. This forcing has no effect in ignore exceptions mode and is superfluous for precise mode.

Instead of using an FPSCR instruction, an execution synchronizing instruction or event can be used to force exceptions and set bits in the FPSCR; however, for the best performance across the widest range of implementations, an FPSCR instruction should be used to achieve these effects.

For the best performance across the widest range of implementations, the following guidelines should be considered:

- If IEEE default results are acceptable, FE0 and FE1 should be cleared (ignore exceptions mode). All FPSCR exception enable bits should be cleared.
- If IEEE default results are unacceptable, an imprecise mode should be used with the FPSCR enable bits set as needed.
- Ignore exceptions mode should not, in general, be used when any FPSCR exception enable bits are set.
- Precise mode may substantially degrade performance in some implementations and should be used only for debugging or other specialized applications.

3.3.6.1 Invalid Operation and Zero Divide Exception Conditions

The flow diagram in Figure 3-23 shows the initial flow for checking floating-point exception conditions (invalid operation and divide by zero conditions). If any of these conditions occur, if FPSCR[FEX] is set (implicitly) and MSR[FE0–FE1] ≠ 00, the processor takes a program exception (floating-point enabled exception type).

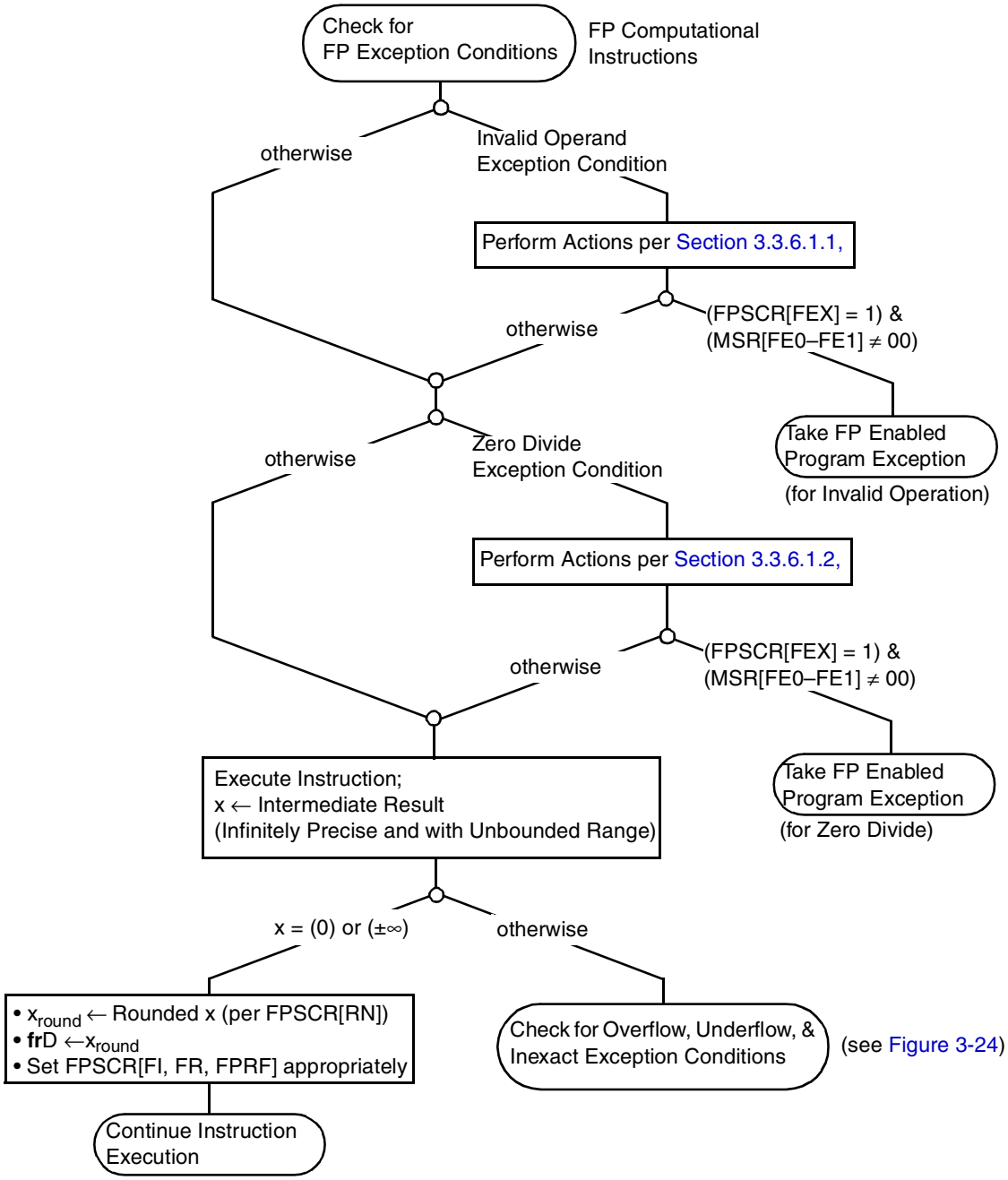


Figure 3-23. Initial Flow for Floating-Point Exception Conditions

See Chapter 6, “Interrupts,” for information about exception handling. The actions performed for floating-point exception conditions are described in the following sections.

3.3.6.1.1 Invalid Operation Exception Condition

An invalid operation exception occurs when an operand is invalid for the specified operation. The invalid operations are as follows:

- Any operation except load, store, move, select, or **mtfsf** on a signaling NaN (SNaN)
- For add or subtract operations, magnitude subtraction of infinities (∞ – ∞)
- Division of infinity by infinity (∞ ÷ ∞)
- Division of zero by zero (0 ÷ 0)
- Multiplication of infinity by zero (∞ * 0)
- Ordered comparison involving a NaN (invalid compare)
- Square root or reciprocal square root of a negative, nonzero number (invalid square root). Note that if the implementation does not support the optional floating-point square root or floating-point reciprocal square root estimate instructions, software can simulate the instruction and set FPSCR[VXSQRT] to reflect the exception.
- Integer convert involving a number that is too large in magnitude to be represented in the target format, or involving an infinity or a NaN (invalid integer convert)

FPSCR[VXSOFT] allows software to cause an invalid operation exception for a condition that is not necessarily associated with the execution of a floating-point instruction. For example, it might be set by a program that computes a square root if the source operand is negative. This allows instructions not implemented in hardware to be emulated.

If an invalid operation occurs or software explicitly requests the exception using FPSCR[VXSOFT], (regardless of the value of FPSCR[VE]), the following occurs:

- One or two invalid operation exception condition bits is set
  FPSCR[VXSNAN] (if SNaN)
  FPSCR[VXISI] (if ∞ – ∞)
  FPSCR[VXIDI] (if ∞ ÷ ∞)
  FPSCR[VXZDZ] (if 0 ÷ 0)
  FPSCR[VXIMZ] (if ∞ * 0)
  FPSCR[VXVC] (if invalid comparison)
  FPSCR[VXSOFT] (if software request)
  FPSCR[VXSQRT] (if invalid square root)
  FPSCR[VXCVI] (if invalid integer convert)
- If the operation is a compare,
  FPSCR[FR, FI, C] are unchanged
  FPSCR[FPCC] is set to reflect unordered
- If software explicitly requests the exception,
  FPSCR[FR, FI, FPRF] are as set by the **mtfsfi**, **mtfsf**, or **mtfsb1** instruction.

Table 3-12 describes additional actions performed that depend on the value of FPSCR[VE].

Table 3-12. Additional Actions Performed for Invalid FP Operations

Invalid Operation	Result Category	Action if FPSCR[VE] = 1	Action if FPSCR[VE] = 0
Arithmetic or floating-point round to single	**fr**D	Unchanged	QNaN
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Set for QNaN	Unchanged
Convert to 64-bit integer (positive number or +∞)	**fr**D[0–63]	Unchanged	Most positive 64-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Set for QNaN	Undefined
Convert to 64-bit integer (negative number, NaN, or –∞)	**fr**D[0–63]	Unchanged	Most negative 64-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Set for QNaN	Undefined
Convert to 32-bit integer (positive number or +∞)	**fr**D[0–31]	Unchanged	Undefined
	frD[32–63]	Unchanged	Most positive 32-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Set for QNaN	Undefined
Convert to 32-bit integer (negative number, NaN, or –∞)	**fr**D[0–31]	Unchanged	Undefined
	frD[32–63]	Unchanged	Most negative 32-bit integer value
	FPSCR[FR, FI]	Cleared	Cleared
	FPSCR[FPRF]	Set for QNaN	Undefined
All cases	FPSCR[FEX]	Implicitly set (causes exception)	Unchanged

3.3.6.1.2 Zero Divide Exception Condition

A zero divide exception condition occurs when a divide instruction is executed with a zero divisor and a finite, nonzero dividend or when an **fres** or **frsqrte** is executed with a zero operand. This condition indicates an exact infinite result from finite operands exception condition corresponding to a mathematical pole (divide or **fres**) or a branch point singularity (**frsqrte**). When a zero divide condition occurs, the following actions are taken:

- Zero divide exception condition bit is set (FPSCR[ZX] = 1).
- FPSCR[FR,FI] are cleared.

Table 3-13 describes additional actions depending on the value of FPSCR[ZE].

Table 3-13. Additional Actions Performed for Zero Divide

Result Category	Action if FPSCR[ZE] = 1	Action if FPSCR[ZE] = 0
frD	Unchanged	±∞ (sign determined by XOR of the signs of the operands)
FPSCR[FEX]	Implicitly set (causes exception)	Unchanged
FPSCR[FPRF]	Unchanged	Set to indicate ±∞

3.3.6.2 Overflow, Underflow, and Inexact Exception Conditions

Overflow, underflow, and inexact exception conditions are detected after the instruction executes and an infinitely precise result with unbounded range is computed. Figure 3-24 shows the flow for the detection of these conditions. It is a continuation of Figure 3-23.

As in the cases of invalid operation, or zero divide conditions, if FPSCR[FEX] is set implicitly as described in Table 3-9 and MSR[FE0,FE1] ≠ 00, the processor takes a program exception (floating-point enabled exception type). Refer to Chapter 6, “Interrupts,” for more information on exception processing. The actions performed for each of these floating-point exception conditions (including the generated result) are described in greater detail in the following sections.

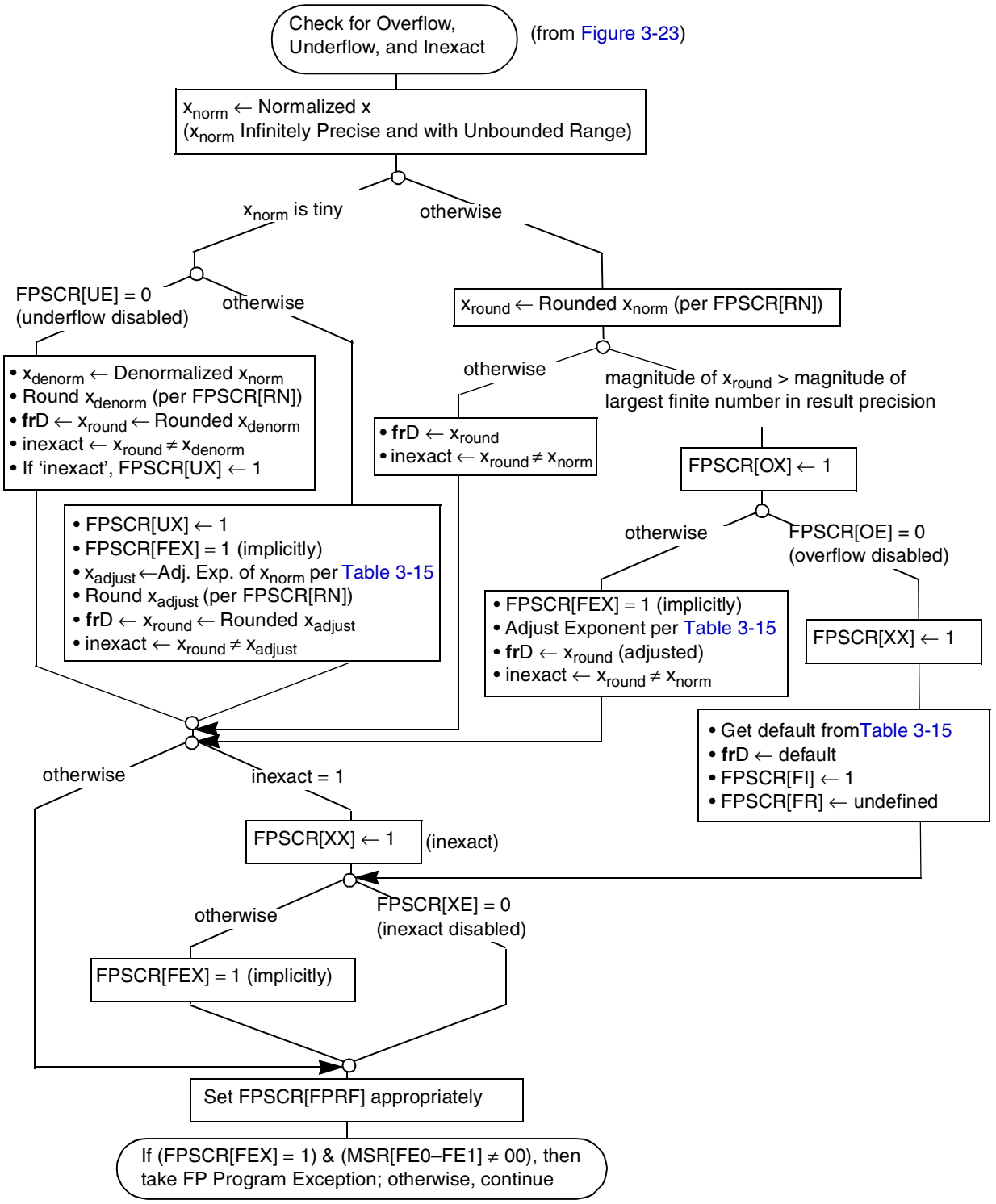


Figure 3-24. Checking of Remaining Floating-Point Exception Conditions

3.3.6.2.1 Overflow Exception Condition

Overflow occurs when the magnitude of what would have been the rounded result (had the exponent range been unbounded) is greater than the magnitude of the largest finite number of the specified result precision. FPSCR[OX] is set, regardless of the FPSCR[OE] value.

Table 3-14 describes additional actions taken that depend on the setting of FPSCR[OE].

Table 3-14. Additional Actions Performed for Overflow Exception Condition

Condition	Result Category	Action if FPSCR[OE] = 1	Action if FPSCR[OE] = 0
Double-precision arithmetic instructions	Exponent of normalized intermediate result	Adjusted by subtracting 1536	—
Single-precision arithmetic and **frsp***x* instruction	Exponent of normalized intermediate result	Adjusted by subtracting 192	—
All cases	**fr**D	Rounded result (with adjusted exponent)	Default result per Table 3-15
	FPSCR[XX]	Set if rounded result differs from intermediate result	Set
	FPSCR[FEX]	Implicitly set (causes exception)	Unchanged
	FPSCR[FPRF]	Set to indicate ±normal number	Set to indicate ±∞ or ±normal number
	FPSCR[FI]	Reflects rounding	Set
	FPSCR[FR]	Reflects rounding	Undefined

When FPSCR[OE] = 0 and an overflow condition occurs, the default result is determined by the rounding mode bit (FPSCR[RN]) and the sign of the intermediate result as shown in Table 3-15.

Table 3-15. Target Result for Overflow Exception Disabled Case

FPSCR[RN]	Sign of Intermediate Result	frD
Round to nearest	Positive	+Infinity
	Negative	–Infinity
Round toward zero	Positive	Format's largest finite positive number
	Negative	Format's most negative finite number
Round toward +infinity	Positive	+Infinity
	Negative	Format's most negative finite number
Round toward –infinity	Positive	Format's largest finite positive number
	Negative	–Infinity

3.3.6.2.2 Underflow Exception Condition

The underflow exception condition is defined separately for the enabled and disabled states:

- Enabled—Underflow occurs when the intermediate result is tiny.
- Disabled—Underflow occurs when the intermediate result is tiny and the rounded result is inexact. In this context, the term ‘tiny’ refers to a floating-point value that is too small to be represented for a particular precision format.

As shown in Figure 3-24, a tiny result is detected before rounding, when a nonzero intermediate result value computed as though it had infinite precision and unbounded exponent range is less in magnitude than the smallest normalized number.

If the intermediate result is tiny and the underflow exception condition enable bit, FPSCR[UE], is zero, the intermediate result is denormalized (see Section 3.3.3, “Normalization and Denormalization”) and rounded (see Section 3.3.5, “Rounding”) before being stored in an FPR. In this case, if the rounding causes the delivered result value to differ from what would have been computed were both the exponent range and precision unbounded (the result is inexact), then underflow occurs and FPSCR[UX] is set.

The actions performed for underflow exception conditions are described in Table 3-16.

Table 3-16. Actions Performed for Underflow Conditions

Condition	Result Category	Action Performed	
		FPSCR[UE] = 1	FPSCR[UE] = 0
Double-precision arithmetic instructions	Exponent of normalized intermediate result	Adjusted by adding 1536	—
Single-precision arithmetic and **frsp***x* instructions	Exponent of normalized intermediate result	Adjusted by adding192	—
All cases	**fr**D	Rounded result (with adjusted exponent)	Denormalized and rounded result
	FPSCR[XX]	Set if rounded result differs from intermediate result	Set if rounded result differs from intermediate result
	FPSCR[UX]	Set	Set only if tiny and inexact after denormalization and rounding
	FPSCR[FPRF]	Set to indicate ±normalized number	Set to indicate ±denormalized number or ±zero
	FPSCR[FEX]	Implicitly set (causes exception)	Unchanged
	FPSCR[FI]	Reflects rounding	Reflects rounding
	FPSCR[FR]	Reflects rounding	Reflects rounding

Note that FPSCR[FR,FI] allow the system floating-point enabled exception error handler, when invoked because of an underflow exception condition, to simulate a trap disabled environment. That is, FR and FI allow the system floating-point enabled exception error handler to unround the result, thus allowing the result to be denormalized.

3.3.6.2.3 Inexact Exception Condition

The inexact exception condition occurs when one of two conditions occur during rounding:

- The rounded result differs from the intermediate result assuming the intermediate result exponent range and precision to be unbounded. (In the case of an enabled overflow or underflow condition, where the exponent of the rounded result is adjusted for those conditions, an inexact condition occurs only if the significand of the rounded result differs from that of the intermediate result.)
- The rounded result overflows and the overflow exception condition is disabled.

When an inexact exception condition occurs, the following actions are taken independently of the setting of the inexact exception condition enable bit of the FPSCR:

- Inexact exception condition bit in the FPSCR is set (FPSCR[XX] = 1).
- The rounded or overflowed result is placed into the target FPR.
- FPSCR[FPRF] is set to indicate the class and sign of the result.

If the inexact exception condition enable bit, FPSCR[XE], is set and an inexact condition exists, then FPSCR[FEX] is implicitly set, causing the processor to take a floating-point enabled program exception. Running with inexact exception conditions enabled may have greater latency than enabling other types of floating-point exception conditions.

Chapter 4
Addressing Modes and Instruction Set Summary

This chapter describes instructions and addressing modes defined by the three levels of the PowerPC architecture—user instruction set architecture (UISA), virtual environment architecture (VEA), and operating environment architecture (OEA). These instructions are divided into the following functional categories:

- Integer instructions—These include arithmetic and logical instructions. For more information, see Section 4.2.1, “Integer Instructions.”
- Floating-point instructions—These include floating-point arithmetic instructions, as well as instructions that affect the floating-point status and control register (FPSCR). For more information, see Section 4.2.2, “Floating-Point Instructions.”
- Load and store instructions—These include integer and floating-point load and store instructions. For more information, see Section 4.2.3, “Load and Store Instructions.”
- Flow control instructions—These include branching instructions, condition register logical instructions, trap instructions, and other instructions that affect the instruction flow. For more information, see Section 4.2.4, “Branch and Flow Control Instructions.”
- Processor control instructions—These instructions are used for synchronizing memory accesses and managing of caches, TLBs, and the segment registers. For more information, see Section 4.2.5, “Processor Control Instructions—UISA.” Section 4.3.1, “Processor Control Instructions—VEA,” and Section 4.4.2, “Processor Control Instructions—OEA.”
- Memory synchronization instructions—These instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms. For more information, see Section 4.2.6, “Memory Synchronization Instructions—UISA,” and Section 4.3.2, “Memory Synchronization Instructions—VEA.”
- Memory control instructions—These include cache management instructions (user-level and supervisor-level), segment register manipulation instructions, and translation lookaside buffer management instructions. For more information, see Section 4.3.3, “Memory Control Instructions—VEA,” and Section 4.4.3, “Memory Control Instructions—OEA.” (Note that user-level and supervisor-level are referred to as problem state and privileged state, respectively, in the architecture specification.)
- External control instructions—These instructions allow a user-level program to communicate with a special-purpose device. For more information, see Section 4.3.4, “External Control Instructions (Optional).”

This grouping of instructions does not necessarily indicate the execution unit that processes a particular instruction or group of instructions within a processor implementation.

U Integer instructions operate on byte, half-word, and word operands. Floating-point instructions operate on single-precision and double-precision floating-point operands. The PowerPC architecture uses instructions that are 4 bytes long and word-aligned. It provides for byte, half-word, and word operand fetches and stores between memory and a set of 32 general-purpose registers (GPRs). It also provides for word and double-word operand fetches and stores between memory and a set of 32 floating-point registers (FPRs). FPRs are 64 bits wide. GPRs are 32 bits wide.

Arithmetic and logical instructions do not read or modify memory. To use the contents of a memory location in a computation and then modify the same or another memory location, the memory contents must be loaded into a register, modified, and then written to the target location using load and store instructions.

The description of each instruction includes the mnemonic and a formatted list of operands. that support the mnemonics and operand lists. To simplify assembly language programming, a set of simplified mnemonics (referred to as extended mnemonics in the architecture specification) and symbols is provided for some of the most frequently-used instructions; see Appendix E, “Simplified Mnemonics for PowerPC Instructions,” for a complete list of simplified mnemonics.

U The instructions are organized by functional categories while maintaining the delineation of the three levels as described in Section 1.1.1, “The Levels of the PowerPC Architecture.”

4.1 Conventions

U This section describes instruction set conventions. Descriptions of computation modes, memory addressing, synchronization, and the interrupt summary follow.

4.1.1 Sequential Execution Model

Processors that implement the PowerPC architecture appear to execute instructions in program order, regardless of asynchronous events or program interrupts. The execution of a sequence of instructions may be interrupted by an interrupt caused by one of the instructions in the sequence, or by an asynchronous event.

For interrupts to the sequential execution model, refer to Chapter 6, “Interrupts.” For information about the synchronization required when using store instructions to access instruction areas of memory, refer to Section 4.2.3.3, “Integer Store Instructions,” and Section 5.2.5.2, “Instruction Cache Instructions.” For information regarding instruction fetching, and for information about guarded memory refer to Section 5.3.1.5, “The Guarded Attribute (G).”

4.1.2 Classes of Instructions

Instructions belong to one of the following three classes:

- Defined
- Illegal
- Reserved

Note that while the definitions of these terms are consistent among these processors, the assignment of these classifications is not.

The class is determined by examining the primary opcode, and the extended opcode if any. If the opcode, or the combination of opcode and extended opcode, is not that of a defined instruction or of a reserved instruction, the instruction is illegal.

In future versions of the architecture, instruction codings that are now illegal may become defined (by being added to the architecture) or reserved (by being assigned to one of the special purposes). Likewise, reserved instructions may become defined.

4.1.2.1 Definition of Boundedly Undefined

The results of executing a given instruction are said to be boundedly undefined if they could have been achieved by executing an arbitrary sequence of instructions, starting in the state the machine was in before executing the given instruction. Boundedly undefined results for a given instruction may vary between implementations and between different executions on the same implementation.

4.1.2.2 Defined Instruction Class

Defined instructions contain all the instructions defined in the UISA, VEA, and OEA. Defined instructions are guaranteed to be supported in all implementations. The only exceptions are optional instructions, as stated in the instruction descriptions in Chapter 8, “Instruction Set.” A processor may invoke the illegal instruction error handler (part of the program interrupt handler) when an unimplemented instruction is encountered so that it may be emulated in software, as required.

A defined instruction can have invalid forms, as described in Section 4.1.2.2.2, “Invalid Instruction Forms.”

4.1.2.2.1 Preferred Instruction Forms

A defined instruction may have an instruction form that is preferred (that is, the instruction will execute in an efficient manner). Any form other than the preferred form will take significantly longer to execute. The following instructions have preferred forms:

- Load/store multiple instructions
- Load/store string instructions
- OR immediate instruction (preferred form of no-op)

4.1.2.2.2 Invalid Instruction Forms

A defined instruction may have an instruction form that is invalid if one or more operands, excluding opcodes, are coded incorrectly in a manner that can be deduced by examining only the instruction encoding (primary and extended opcodes). Attempting to execute an invalid form of an instruction either invokes the illegal instruction error handler (a program interrupt) or yields boundedly-undefined results. See Chapter 8, “Instruction Set,” for individual instruction descriptions.

Invalid forms result when a bit or operand is coded incorrectly, for example, or when a reserved bit (shown as ‘0’) is coded as ‘1’.

The following instructions have invalid forms identified in their individual instruction descriptions:

- Branch conditional instructions
- Load/store with update instructions
- Load multiple instructions
- Load string instructions
- Integer compare instructions (in 32-bit implementations only)
- Load/store floating-point with update instructions

4.1.2.2.3 Optional Instructions

A defined instruction may be optional. The optional instructions fall into the following categories:

- General-purpose instructions—**fsqrt** and **fsqrts**
- Graphics instructions—**fres**, **frsqrte**, and **fsel**
- External control instructions—**eciwx** and **ecowx**
- Lookaside buffer management instructions—**tlbia**, **tlbie**, and **tlbsync** (with conditions, see Chapter 8, “Instruction Set,” for more information)

Note that the **stfiwx** instruction is defined as optional by the architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent processors.

Also, note that additional categories may be defined in future implementations. If an implementation claims to support a given category, it implements all the instructions in that category.

Any attempt to execute an optional instruction that is not provided by the implementation causes the illegal instruction error handler to be invoked. Exceptions to this rule are stated in the instruction descriptions found in Chapter 8, “Instruction Set.”

4.1.2.3 Illegal Instruction Class

Illegal instructions can be grouped into the following categories:

- Instructions that are not implemented in the architecture. These opcodes are available for future extensions of the architecture; that is, future versions of the architecture may define any of these instructions to perform new functions. The following primary opcodes are defined as illegal but may be used in future extensions to the architecture:

  1, 4, 5, 6, 56, 57, 60, 61
- Instructions that are implemented in the architecture but are not implemented in a specific implementation. For example, optional instructions that are not implemented.

  The following primary opcodes are defined for 64-bit implementations only and are illegal on 32-bit implementations:

  2, 30, 58, 62
- All unused extended opcodes are illegal. The unused extended opcodes can be determined from information in Section A.4, “Instructions Sorted by Opcode (Binary),” and Section 4.1.2.4, “Reserved Instructions.” Notice that extended opcodes for instructions that are defined only for

64-bit implementations are illegal in 32-bit implementations. The following primary opcodes have unused extended opcodes.

19, 31, 59, 63 (primary opcodes 30 and 62 are illegal for 32-bit implementations, but as 64-bit opcodes they have some unused extended opcodes)

- An instruction consisting entirely of zeros is guaranteed to be an illegal instruction. This increases the probability that an attempt to execute data or uninitialized memory invokes the illegal instruction error handler (a program interrupt). Note that if only the primary opcode consists of all zeros, the instruction is considered a reserved instruction, as described in Section 4.1.2.4, “Reserved Instructions.”

An attempt to execute an illegal instruction invokes the illegal instruction error handler (a program interrupt) but has no other effect. See Section 6.5.7, “Program Interrupt (0x00700).”

Except for the instruction consisting entirely of binary zeros, the illegal instructions are available for further additions to the architecture.

4.1.2.4 Reserved Instructions

Reserved instructions are allocated to specific implementation-dependent purposes not defined by the architecture. An attempt to execute an unimplemented reserved instruction invokes the illegal instruction program interrupt. See Section 6.5.7, “Program Interrupt (0x00700).”

The following types of instructions are reserved:

- POWER architecture instructions not included in the architecture.
- Implementation-specific instructions not defined in the UISA, VEA, or OEA, including those used to conform to the architecture specifications (for example, Load Data TLB Entry (**tlbld**) and Load Instruction TLB Entry (**tlbli**) instructions implemented in several processors).
- The instruction with primary opcode 0 when the instruction does not consist entirely of binary zeros

4.1.3 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next sequential instruction.

4.1.3.1 Memory Operands

Bytes in memory are numbered consecutively starting with zero. Each number is the address of the corresponding byte.

Memory operands may be bytes, half words, words, or double words, or, for the load/store multiple and load/store string instructions, a sequence of bytes or words. The address of a memory operand is the address of its first byte (that is, of its lowest-numbered byte). Operand length is implicit for each instruction. The architecture supports both big-endian and little-endian byte ordering. The default byte and bit ordering is big-endian; see Section 3.1.2, “Byte Ordering,” for more information.

The operand of a single-register memory access instruction has a natural alignment boundary equal to the operand length. In other words, the “natural” address of an operand is an integral multiple of the operand length. A memory operand is said to be aligned if it is aligned at its natural boundary; otherwise it is misaligned. For a detailed discussion about memory operands, see Chapter 3, “Operand Conventions.”

4.1.3.2 Effective Address Calculation

An effective address (EA) is the 32-bit sum computed by the processor when executing a memory access or branch instruction or when fetching the next sequential instruction. For a memory access instruction, if the sum of the effective address and the operand length exceeds the maximum effective address, the memory operand is considered to wrap around from the maximum effective address through effective address 0, as described in the following paragraphs.

Effective address computations for both data and instruction accesses use 32-bit unsigned binary arithmetic. A carry from bit 0 is ignored.

U In all implementations, the three low-order bits of the calculated effective address may be modified by the processor before accessing memory if the system is operating in little-endian mode. See Section 3.1.2, “Byte Ordering.”

Load and store operations have three categories of effective address generation that depend on the operands specified:

- Register indirect with immediate index mode
- Register indirect with index mode
- Register indirect mode

  See Section 4.2.3.1, “Integer Load and Store Address Generation,” for a detailed description of effective address generation for load and store operations.

Branch instructions have three categories of effective address generation:

- Immediate addressing
- Link register indirect
- Count register indirect

See Section 4.2.4.1, “Branch Instruction Address Calculation.”

Branch instructions can optionally load the LR with the next sequential instruction address (current instruction address + 4).

4.1.4 Synchronizing Instructions

O The synchronization described in this section refers to the state of activities within the processor that is performing the synchronization. Detailed information is provided in Section 6.2.2, “Context Synchronization,” about other conditions that can cause context and execution synchronization.

4.1.4.1 Context Synchronizing Instructions

The System Call (**sc**), Return from Interrupt (**rfi**), and Instruction Synchronize (**isync**) instructions perform context synchronization by allowing previously issued instructions to complete before performing a context switch. Execution of one of these instructions ensures the following:

- No higher priority interrupt exists (**sc**) and instruction dispatching is halted.
- All previous instructions have completed to a point where they can no longer cause an interrupt.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The instructions following an **sc**, **rfi**, or **isync** instruction execute in the context established by these instructions.

4.1.4.2 Execution Synchronizing Instructions

An instruction is execution synchronizing if it satisfies the conditions of the first two items described above for context synchronization. The **sync** instruction is treated like **isync** with respect to the second item above (that is, the conditions described in the second item apply to the completion of **sync**). The **sync** and **mtmsr** instructions are examples of execution-synchronizing instructions.

All context-synchronizing instructions are execution-synchronizing. Unlike a context synchronizing operation, an execution synchronizing instruction need not ensure that the instructions following it execute in the context established by that instruction. This new context becomes effective sometime after the execution synchronizing instruction completes and before or at a subsequent context synchronizing operation.

4.1.5 Interrupt Summary

The interrupt mechanism handles system functions and error conditions in an orderly way. The interrupt model is defined by the OEA. There are two kinds of interrupts—those caused directly by the execution of an instruction and those caused by an asynchronous event. Either may cause components of the system software to be invoked.

Interrupts can be caused directly by the execution of an instruction as follows:

- An attempt to execute an illegal instruction causes the illegal instruction (program interrupt) error handler to be invoked. An attempt by a user-level program to execute the supervisor-level instructions listed below causes the privileged instruction (program interrupt) handler to be invoked.

  The architecture provides the following supervisor-level instructions: **dcbi**, **mfmsr**, **mfspr**, **mfsr**, **mfsrin**, **mtmsr**, **mtspr**, **mtsr**, **mtsrin**, **rfi**, **tlbia**, **tlbie**, and **tlbsync** (defined by OEA). Note that the privilege level of the **mfspr** and **mtspr** instructions depends on the access level of the specified SPR. See Figure 2-11.
- The execution of a defined instruction using an invalid form causes either the illegal instruction error handler or the privileged instruction handler to be invoked.
- The execution of an optional instruction that is not provided by the implementation causes the illegal instruction error handler to be invoked.

- An attempt to access memory in a manner that violates memory protection, or an attempt to access memory that is not available (page fault), causes the data storage interrupt or instruction storage interrupt handler to be invoked.
- An attempt to access memory with an effective address alignment that is invalid for the instruction causes the alignment interrupt handler to be invoked.
- The execution of an **sc** instruction permits a program to call on the system to perform a service, by causing a system call interrupt handler to be invoked.
- The execution of a trap instruction invokes the program interrupt trap handler.
- The execution of a floating-point instruction when floating-point instructions are disabled invokes the floating-point unavailable interrupt handler.
- The execution of an instruction that causes a floating-point interrupt that is enabled invokes the floating-point enabled interrupt handler.
- The execution of a floating-point instruction that requires system software assistance causes the floating-point assist interrupt handler to be invoked. The conditions under which such software assistance is required are implementation-dependent.

Interrupts caused by asynchronous events are described in Chapter 6, “Interrupts.”

4.1.6 Recommended Simplified Mnemonics

To simplify assembly language programs, a set of simplified mnemonics is provided for some of the most frequently used operations (such as no-op, load immediate, load address, move register, and complement register). Assemblers should provide the simplified mnemonics listed in Section E.9, “Recommended Simplified Mnemonics.” Programs written to be portable across the various assemblers for the architecture should not assume the existence of mnemonics not described in this document.

For a complete list of guaranteed simplified mnemonics, see Appendix E, “Simplified Mnemonics for PowerPC Instructions.” Note that some assemblers may define additional simplified mnemonics not described here.

4.2 UISA Instructions

The user instruction set architecture (UISA) includes the base user-level instruction set (excluding a few user-level cache-control, synchronization, and time base instructions), user-level registers, programming model, data types, and addressing modes. This section discusses the instructions defined in the UISA.

4.2.1 Integer Instructions

The sections included here describe the instructions by functionality, as follows:

- Section 4.2.1.1, “Integer Arithmetic Instructions”
- Section 4.2.1.2, “Integer Compare Instructions”
- Section 4.2.1.3, “Integer Logical Instructions”
- Section 4.2.1.4, “Integer Rotate and Shift Instructions”

Integer instructions use the content of the GPRs as source operands and place results into GPRs. Integer arithmetic, shift, rotate, and string move instructions may update or read values from the XER, and the condition register (CR) fields may be updated if the Rc bit of the instruction is set.

These instructions treat the source operands as signed integers unless the instruction is explicitly identified as performing an unsigned operation. For example, Multiply High-Word Unsigned (**mulhwu**) and Divide Word Unsigned (**divwu**) instructions interpret both operands as unsigned integers.

The integer instructions that are coded to update the condition register, and the integer arithmetic instruction, **addic.**, set CR bits 0–3 (CR0) to characterize the result of the operation. CR0 is set to reflect a signed comparison of the result to zero.

The integer arithmetic instructions, **addic**, **addic.**, **subfic**, **addc**, **subfc**, **adde**, **subfe**, **addme**, **subfme**, **addze**, and **subfze**, always set the XER bit, CA, to reflect the carry out of bit 0. Integer arithmetic instructions with the overflow enable (OE) bit set in the instruction encoding (instructions with o suffix) cause the XER[SO] and XER[OV] to reflect an overflow of the result. Except for the multiply low and divide instructions, these integer arithmetic instructions reflect the overflow of the result.

Instructions that select the overflow option (enable XER[OV]) or that set the XER carry bit (CA) may delay the execution of subsequent instructions.

Unless otherwise noted, when CR0 and the XER are set, they reflect the value placed in the target register.

4.2.1.1 Integer Arithmetic Instructions

Table 4-1 lists the integer arithmetic instructions.

Table 4-1. Integer Arithmetic Instructions

Name	Mnemonic	Syntax	Operation
Add Immediate	**addi**	**r**D,**r**A,SIMM	The sum (**r**A|0) + SIMM is placed into **r**D.
Add Immediate Shifted	**addis**	**r**D,**r**A,SIMM	The sum (**r**A|0) + (SIMM || 0x0000) is placed into **r**D.
Add	**add** **add.** **addo** **addo.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) is placed into **r**D. **add** Add **add.** Add with CR Update. The dot suffix enables the update of the CR. **addo** Add with Overflow Enabled. The o suffix enables XER[OV]. **addo.** Add with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Subtract From	**subf** **subf.** **subfo** **subfo.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) +1 is placed into **r**D. **subf** Subtract From **subf.** Subtract from with CR Update. The dot suffix enables the update of the CR. **subfo** Subtract from with Overflow Enabled. The o suffix enables XER[OV]. **subfo.** Subtract from with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Add Immediate Carrying	**addic**	**r**D,**r**A,SIMM	The sum (**r**A) + SIMM is placed into **r**D.

Table 4-1. Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax	Operation
Add Immediate Carrying and Record	**addic.**	**r**D,**r**A,SIMM	The sum (**r**A) + SIMM is placed into **r**D. The CR is updated.
Subtract from Immediate Carrying	**subfic**	**r**D,**r**A,SIMM	The sum ¬ (**r**A) + SIMM + 1 is placed into **r**D.
Add Carrying	**addc** **addc.** **addco** **addco.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) is placed into **r**D. **addc** Add Carrying **addc.** Add Carrying with CR Update. The dot suffix enables the update of the CR. **addco** Add Carrying with Overflow Enabled. The o suffix enables XER[OV]. **addco.** Add Carrying with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Subtract from Carrying	**subfc** **subfc.** **subfco** **subfco.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D. **subfc** Subtract from Carrying **subfc.** Subtract from Carrying with CR Update. The dot suffix enables the update of the CR. **subfco** Subtract from Carrying with Overflow. The o suffix enables XER[OV]. **subfco.** Subtract from Carrying with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Add Extended	**adde** **adde.** **addeo** **addeo.**	**r**D,**r**A,**r**B	The sum (**r**A) + (**r**B) + XER[CA] is placed into **r**D. **adde** Add Extended **adde.** Add Extended with CR Update. The dot suffix enables the update of the CR. **addeo** Add Extended with Overflow. The o suffix enables XER[OV]. **addeo.** Add Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Subtract from Extended	**subfe** **subfe.** **subfeo** **subfeo.**	**r**D,**r**A,**r**B	The sum ¬ (**r**A) + (**r**B) + XER[CA] is placed into **r**D. **subfe** Subtract from Extended **subfe.** Subtract from Extended with CR Update. The dot suffix enables the update of the CR. **subfeo** Subtract from Extended with Overflow. The o suffix enables XER[OV]. **subfeo.** Subtract from Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow (OV) bit in the XER.
Add to Minus One Extended	**addme** **addme.** **addmeo** **addmeo.**	**r**D,**r**A	The sum (**r**A) + XER[CA] added to 0xFFFF_FFFF is placed into **r**D. **addme** Add to Minus One Extended **addme.** Add to Minus One Extended with CR Update. The dot suffix enables the update of the CR. **addmeo** Add to Minus One Extended with Overflow. The o suffix enables XER[OV]. **addmeo.** Add to Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables the overflow (OV) bit in the XER.

Table 4-1. Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax	Operation
Subtract from Minus One Extended	**subfme** **subfme.** **subfmeo** **subfmeo.**	**r**D,**r**A	The sum ¬ (**r**A) + XER[CA] added to 0xFFFF_FFFF is placed into **r**D. **subfme** Subtract from Minus One Extended **subfme.** Subtract from Minus One Extended with CR Update. The dot suffix enables the update of the CR. **subfmeo** Subtract from Minus One Extended with Overflow. The o suffix enables XER[OV]. **subfmeo.** Subtract from Minus One Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Add to Zero Extended	**addze** **addze.** **addzeo** **addzeo.**	**r**D,**r**A	The sum (**r**A) + XER[CA] is placed into **r**D. **addze** Add to Zero Extended **addze.** Add to Zero Extended with CR Update. The dot suffix enables the update of the CR. **addzeo** Add to Zero Extended with Overflow. The o suffix enables XER[OV]. **addzeo.** Add to Zero Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Subtract from Zero Extended	**subfze** **subfze.** **subfzeo** **subfzeo.**	**r**D,**r**A	The sum ¬ (**r**A) + XER[CA] is placed into **r**D. **subfze** Subtract from Zero Extended **subfze.** Subtract from Zero Extended with CR Update. The dot suffix enables the update of the CR. **subfzeo** Subtract from Zero Extended with Overflow. The o suffix enables XER[OV]. **subfzeo.** Subtract from Zero Extended with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Negate	**neg** **neg.** **nego** **nego.**	**r**D,**r**A	The sum ¬ (**r**A) + 1 is placed into **r**D. **neg** Negate **neg.** Negate with CR Update. The dot suffix enables the update of the CR. **nego** Negate with Overflow. The o suffix enables XER[OV]. **nego.** Negate with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Multiply Low Immediate	**mulli**	**r**D,**r**A,SIMM	The low-order 32 bits of the product (**r**A) * SIMM are placed into **r**D.
Multiply Low	**mullw** **mullw.** **mullwo** **mullwo.**	**r**D,**r**A,**r**B	The 32-bit product (**r**A) * (**r**B) is placed into register **r**D. **mullw** Multiply Low **mullw.** Multiply Low with CR Update. The dot suffix enables the update of the CR. **mullwo** Multiply Low with Overflow. The o suffix enables XER[OV]. **mullwo.** Multiply Low with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Multiply High Word	**mulhw** **mulhw.**	**r**D,**r**A,**r**B	The contents of **r**A and **r**B are interpreted as 32-bit signed integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into **r**D. **mulhw** Multiply High Word **mulhw.** Multiply High Word with CR Update. The dot suffix enables the update of the CR.
Multiply High Word Unsigned	**mulhwu** **mulhwu.**	**r**D,**r**A,**r**B	The contents of **r**A and of **r**B are interpreted as 32-bit unsigned integers. The 64-bit product is formed. The high-order 32 bits of the 64-bit product are placed into **r**D. **mulhwu** Multiply High Word Unsigned **mulhwu.** Multiply High Word Unsigned with CR Update. The dot suffix enables the update of the CR.

Table 4-1. Integer Arithmetic Instructions (continued)

Name	Mnemonic	Syntax	Operation
Divide Word	**divw** **divw.** **divwo** **divwo.**	**r**D,**r**A,**r**B	The dividend is the signed value of **r**A. The divisor is the signed value of **r**B. The quotient is placed into **r**D. The remainder is not supplied as a result. **divw** Divide Word **divw.** Divide Word with CR Update. The dot suffix enables the update of the CR. **divwo** Divide Word with Overflow. The o suffix enables XER[OV]. **divwo.** Divide Word with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].
Divide Word Unsigned	**divwu** **divwu.** **divwuo** **divwuo.**	**r**D,**r**A,**r**B	The dividend is the zero-extended value in **r**A. The divisor is the zero-extended value in **r**B. The quotient is placed into **r**D. The remainder is not supplied as a result. **divwu** Divide Word Unsigned **divwu.** Divide Word Unsigned with CR Update. The dot suffix enables the update of the CR. **divwuo** Divide Word Unsigned with Overflow. The o suffix enables XER[OV]. **divwuo.** Divide Word Unsigned with Overflow and CR Update. The o. suffix enables the update of the CR and enables XER[OV].

Although there is no Subtract Immediate instruction, its effect can be achieved by using an **addi** instruction with the immediate operand negated. Simplified mnemonics are provided that include this negation. The **subf** instructions subtract the second operand (**r**A) from the third operand (**r**B). Simplified mnemonics are provided in which the third operand is subtracted from the second operand. See Appendix E, “Simplified Mnemonics for PowerPC Instructions,” for examples.

4.2.1.2 Integer Compare Instructions

The integer compare instructions algebraically or logically compare the contents of register **r**A with either the zero-extended value of the UIMM operand, the sign-extended value of the SIMM operand, or the contents of register **r**B. The comparison is signed for the **cmpi** and **cmp** instructions, and unsigned for the **cmpli** and **cmpl** instructions. Table 4-2 summarizes the integer compare instructions.

For 32-bit implementations, the L field must be cleared, otherwise the instruction form is invalid.

The integer compare instructions (Table 4-2) set one of the leftmost three bits of the designated CR field and clear the other two. XER[SO] is copied into bit 3 of the CR field.

Table 4-2. Integer Compare Instructions

Name	Mnemonic	Syntax	Operation
Compare Immediate	**cmpi**	**crf**D,L,**r**A,SIMM	The value in register **r**A is compared with the sign-extended value of the SIMM operand, treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare	**cmp**	**crf**D,L,**r**A,**r**B	The value in register **r**A is compared with the value in register **r**B, treating the operands as signed integers. The result of the comparison is placed into the CR field specified by operand **crf**D.

Table 4-2. Integer Compare Instructions (continued)

Name	Mnemonic	Syntax	Operation
Compare Logical Immediate	**cmpli**	**crf**D,L,**r**A,UIMM	The value in register **r**A is compared with 0x0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand **crf**D.
Compare Logical	**cmpl**	**crf**D,L,**r**A,**r**B	The value in register **r**A is compared with the value in register **r**B, treating the operands as unsigned integers. The result of the comparison is placed into the CR field specified by operand **crf**D.

The **crf**D operand can be omitted if the result of the comparison is to be placed in CR0. Otherwise the target CR field must be specified in the instruction **crf**D field, using an explicit field number.

For information on simplified mnemonics for the integer compare instructions see Appendix E, "Simplified Mnemonics for PowerPC Instructions."

4.2.1.3 Integer Logical Instructions

The logical instructions shown in Table 4-3 perform bit-parallel operations on 32-bit operands. Logical instructions with the CR updating enabled (uses dot suffix) and instructions **andi.** and **andis.** set CR field CR0 (bits 0 to 2) to characterize the result of the logical operation. Logical instructions without CR update and the remaining logical instructions do not modify the CR. Logical instructions do not affect XER[SO,OV,CA].

See Appendix E, "Simplified Mnemonics for PowerPC Instructions," for simplified mnemonic examples for integer logical operations.

Table 4-3. Integer Logical Instructions

Name	Mnemonic	Syntax	Operation
AND Immediate	**andi.**	**r**A,**r**S,UIMM	The contents of **r**S are ANDed with 0x0000 || UIMM and the result is placed into **r**A. The CR is updated.
AND Immediate Shifted	**andis.**	**r**A,**r**S,UIMM	The content of **r**S are ANDed with UIMM || 0x0000 and the result is placed into **r**A. The CR is updated.
OR Immediate	**ori**	**r**A,**r**S,UIMM	The contents of **r**S are ORed with 0x0000 || UIMM and the result is placed into **r**A. The preferred no-op is **ori 0,0,0**
OR Immediate Shifted	**oris**	**r**A,**r**S,UIMM	The contents of **r**S are ORed with UIMM || 0x0000 and the result is placed into **r**A.
XOR Immediate	**xori**	**r**A,**r**S,UIMM	The contents of **r**S are XORed with 0x0000 || UIMM and the result is placed into **r**A.
XOR Immediate Shifted	**xoris**	**r**A,**r**S,UIMM	The contents of **r**S are XORed with UIMM || 0x0000 and the result is placed into **r**A.

Table 4-3. Integer Logical Instructions (continued)

Name	Mnemonic	Syntax	Operation
AND	**and** **and.**	**r**A,**r**S,**r**B	The contents of **r**S are ANDed with the contents of register **r**B and the result is placed into **r**A. **and** AND **and.** AND with CR Update**.** The dot suffix enables the update of the CR.
OR	**or** **or.**	**r**A,**r**S,**r**B	The contents of **r**S are ORed with the contents of **r**B and the result is placed into **r**A. **or** OR **or.** OR with CR Update**.** The dot suffix enables the update of the CR.
XOR	**xor** **xor.**	**r**A,**r**S,**r**B	The contents of **r**S are XORed with the contents of **r**B and the result is placed into **r**A. **xor** XOR **xor.** XOR with CR Update**.** The dot suffix enables the update of the CR.
NAND	**nand** **nand.**	**r**A,**r**S,**r**B	The contents of **r**S are ANDed with the contents of **r**B and the one's complement of the result is placed into **r**A. **nand** NAND **nand.** NAND with CR Update**.** The dot suffix enables the update of CR. Note that **nand***x*, with **r**S = **r**B, can be used to obtain the one's complement.
NOR	**nor** **nor.**	**r**A,**r**S,**r**B	The contents of **r**S are ORed with the contents of **r**B and the one's complement of the result is placed into **r**A. **nor** NOR **nor.** NOR with CR Update**.** The dot suffix enables the update of the CR. Note that **nor***x*, with **r**S = **r**B, can be used to obtain the one's complement.
Equivalent	**eqv** **eqv.**	**r**A,**r**S,**r**B	The contents of **r**S are XORed with the contents of **r**B and the complemented result is placed into **r**A. **eqv** Equivalent **eqv.** Equivalent with CR Update**.** The dot suffix enables the update of the CR.
AND with Complement	**andc** **andc.**	**r**A,**r**S,**r**B	The contents of **r**S are ANDed with the one's complement of the contents of **r**B and the result is placed into **r**A. **andc** AND with Complement **andc.** AND with Complement with CR Update**.** The dot suffix enables the update of the CR.
OR with Complement	**orc** **orc.**	**r**A,**r**S,**r**B	The contents of **r**S are ORed with the complement of the contents of **r**B and the result is placed into **r**A. **orc** OR with Complement **orc.** OR with Complement with CR Update**.** The dot suffix enables the update of the CR.
Extend Sign Byte	**extsb** **extsb.**	**r**A,**r**S	The contents of the low-order 8 bits of **r**S are placed into the low-order 8 bits of **r**A. Bit 24 of **r**S is placed into the remaining high-order bits of **r**A. **extsb** Extend Sign Byte **extsb.** Extend Sign Byte with CR Update**.** The dot suffix enables the update of the CR.

Table 4-3. Integer Logical Instructions (continued)

Name	Mnemonic	Syntax	Operation
Extend Sign Half Word	**extsh** **extsh.**	**r**A,**r**S	The contents of the low-order 16 bits of **r**S are placed into the low-order 16 bits of **r**A. Bit 16 of **r**S is placed into the remaining high-order bits of **r**A. **extsh** Extend Sign Half Word **extsh.** Extend Sign Half Word with CR Update**.** The dot suffix enables the update of the CR.
Count Leading Zeros Word	**cntlzw** **cntlzw.**	**r**A,**r**S	A count of the number of consecutive zero bits starting at bit 0 of **r**S is placed into **r**A. This number ranges from 0 to 32, inclusive. If Rc = 1 (dot suffix), LT is cleared in CR0. **cntlzw** Count Leading Zeros Word **cntlzw.** Count Leading Zeros Word with CR Update**.** The dot suffix enables the update of the CR.

4.2.1.4 Integer Rotate and Shift Instructions

Rotation operations are performed on data from a GPR, and the result, or a portion of the result, is returned to a GPR. The rotation operations rotate a 32-bit quantity left by a specified number of bit positions. Bits that exit from position 0 enter at position 31.

The rotate and shift instructions employ a mask generator. The mask is 32 bits long and consists of ‘1’ bits from a start bit, Mstart, through and including a stop bit, Mstop, and ‘0’ bits elsewhere. The values of Mstart and Mstop range from 0 to 31. If Mstart > Mstop, the ‘1’ bits wrap around from position 31 to position 0. Thus the mask is formed as follows:

```
if Mstart ≤ Mstop then
         mask[mstart:mstop] = ones
         mask[all other bits] = zeros
else
         mask[mstart:31] = ones
         mask[0:mstop] = ones
         mask[all other bits] = zeros
```

It is not possible to specify an all-zero mask. The use of the mask is described in the following sections.

If CR updating is enabled, rotate and shift instructions set CR0[0–2] according to the contents of **r**A at the completion of the instruction. Rotate and shift instructions do not change the values of XER[OV] and XER[SO] bits. Rotate and shift instructions, except algebraic right shifts, do not change the XER[CA] bit.

See Appendix E, “Simplified Mnemonics for PowerPC Instructions,” for a complete list of simplified mnemonics that allows simpler coding of often-used functions such as clearing the leftmost or rightmost bits of a register, left justifying or right justifying an arbitrary field, and simple rotates and shifts.

4.2.1.4.1 Integer Rotate Instructions

Integer rotate instructions rotate the contents of a register. The result of the rotation is either inserted into the target register under control of a mask (if a mask bit is 1 the associated bit of the rotated data is placed into the target register, and if the mask bit is 0 the associated bit in the target register is unchanged), or ANDed with a mask before being placed into the target register.

Rotate left instructions allow right-rotation of the contents of a register to be performed by a left-rotation 32 – *n*, where *n* is the number of bits by which to rotate right.

The integer rotate instructions are summarized in Table 4-4.

Table 4-4. Integer Rotate Instructions

Name	Mnemonic	Syntax	Operation
Rotate Left Word Immediate then AND with Mask	**rlwinm** **rlwinm.**	**r**A,**r**S,SH,MB,ME	The contents of register **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having 1 bits from the bit specified by operand MB through the bit specified by operand ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into register **r**A. **rlwinm** Rotate Left Word Immediate then AND with Mask **rlwinm.** Rotate Left Word Immediate then AND with Mask with CR Update. The dot suffix enables the update of the CR.
Rotate Left Word then AND with Mask	**rlwnm** **rlwnm.**	**r**A,**r**S,**r**B,MB,ME	The contents of **r**S are rotated left by the number of bits specified by operand in the low-order five bits of **r**B. A mask is generated having 1 bits from the bit specified by operand MB through the bit specified by operand ME and 0 bits elsewhere. The rotated word is ANDed with the generated mask and the result is placed into **r**A. **rlwnm** Rotate Left Word then AND with Mask **rlwnm.** Rotate Left Word then AND with Mask with CR Update. The dot suffix enables the update of the CR.
Rotate Left Word Immediate then Mask Insert	**rlwimi** **rlwimi.**	**r**A,**r**S,SH,MB,ME	The contents of **r**S are rotated left by the number of bits specified by operand SH. A mask is generated having 1 bits from the bit specified by operand MB through the bit specified by operand ME and 0 bits elsewhere. The rotated word is inserted into **r**A under control of the generated mask. **rlwimi** Rotate Left Word Immediate then Mask **rlwimi.** Rotate Left Word Immediate then Mask Insert with CR Update. The dot suffix enables the update of the CR.

4.2.1.4.2 Integer Shift Instructions

The integer shift instructions perform left and right shifts. Immediate-form logical (unsigned) shift operations are obtained by specifying masks and shift values for certain rotate instructions. Simplified mnemonics (shown in Appendix E, “Simplified Mnemonics for PowerPC Instructions”) are provided to make coding of such shifts simpler and easier to understand.

Any shift right algebraic instruction, followed by **addze**, can be used to divide quickly by 2^n. The setting of XER[CA] by the shift right algebraic instruction is independent of mode.

Multiple-precision shifts can be programmed as shown in Appendix B, “Multiple-Precision Shifts.”

The integer shift instructions are summarized in Table 4-5.

Table 4-5. Integer Shift Instructions

Name	Mnemonic	Syntax	Operation
Shift Left Word	**slw** **slw.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted left the number of bits specified by operand in the low-order six bits of **r**B. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into **r**A. **slw** Shift Left Word **slw.** Shift Left Word with CR Update. The dot suffix enables the update of the CR.
Shift Right Word	**srw** **srw.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted right the number of bits specified by the low-order 6 **r**B bits. Bits shifted out of position 31 are lost. Zeros are supplied to the vacated positions on the left. The 32-bit result is placed into **r**A. **srw** Shift Right Word **srw.** Shift Right Word with CR Update. The dot suffix enables the update of the CR.
Shift Right Algebraic Word Immediate	**srawi** **srawi.**	**r**A,**r**S,SH	The contents of **r**S are shifted right the number of bits specified by operand SH. Bits shifted out of position 31are lost. The result is sign extended and placed into **r**A. **srawi** Shift Right Algebraic Word Immediate **srawi.** Shift Right Algebraic Word Immediate with CR Update. The dot suffix enables the update of the CR.
Shift Right Algebraic Word	**sraw** **sraw.**	**r**A,**r**S,**r**B	The contents of **r**S are shifted right the number of bits specified by the low-order six bits of **r**B. Bits shifted out of position 31 are lost. The result is placed into **r**A. **sraw** Shift Right Algebraic Word **sraw.** Shift Right Algebraic Word with CR Update. The dot suffix enables the update of the CR.

4.2.2 Floating-Point Instructions

This sections included here describes the floating-point instructions by functionality, as follows:

- Section 4.2.2.1, “Floating-Point Arithmetic Instructions”
- Section 4.2.2.2, “Floating-Point Multiply-Add Instructions”
- Section 4.2.2.3, “Floating-Point Rounding and Conversion Instructions”
- Section 4.2.2.4, “Floating-Point Compare Instructions”
- Section 4.2.2.5, “Floating-Point Status and Control Register Instructions”
- Section 4.2.2.6, “Floating-Point Move Instructions”

Note that MSR[FP] must be set for any of these instructions (including the floating-point loads and stores) to be executed. If MSR[FP] = 0 when any floating-point instruction is attempted, the floating-point unavailable interrupt is taken (see Section 6.5.8, “Floating-Point Unavailable Interrupt (0x00800)”). Section 4.2.3, “Load and Store Instructions,” describes floating-point loads and stores.

The architecture supports the IEEE-754 floating-point standard, but requires software support to conform with that standard. Floating-point operations conform to the standard, except for operations performed with the **fmadd**, **fres**, **fsel**, and **frsqrte** instructions, or if software sets the non-IEEE mode bit, FPSCR[NI]. Section 3.3, “Floating-Point Execution Models—UISA,” gives detailed information about the floating-point formats and interrupt conditions. Also, see Appendix C, “Floating-Point Models.”

4.2.2.1 Floating-Point Arithmetic Instructions

The floating-point arithmetic instructions are summarized in Table 4-6.

Table 4-6. Floating-Point Arithmetic Instructions

Name	Mnemonic	Syntax	Operation
Floating Add (Double-Precision)	**fadd** **fadd.**	**fr**D,**fr**A,**fr**B	The floating-point operand in **fr**A is added to the floating-point operand in **fr**B. If the most significant bit of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D. **fadd** Floating Add (Double-Precision) **fadd.** Floating Add (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Add Single	**fadds** **fadds.**	**fr**D,**fr**A,**fr**B	The floating-point operand in **fr**A is added to the floating-point operand in **fr**B. If the most significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D. **fadds** Floating Add Single **fadds.** Floating Add Single with CR Update. The dot suffix enables the update of the CR.
Floating Subtract (Double-Precision)	**fsub** **fsub.**	**fr**D,**fr**A,**fr**B	The floating-point operand in **fr**B is subtracted from the floating-point operand in **fr**A. If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D. **fsub** Floating Subtract (Double-Precision) **fsub.** Floating Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Subtract Single	**fsubs** **fsubs.**	**fr**D,**fr**A,**fr**B	The floating-point operand in **fr**B is subtracted from the floating-point operand in **fr**A. If the most significant bit of the resultant significand is not 1, the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D. **fsubs** Floating Subtract Single **fsubs.** Floating Subtract Single with CR Update. The dot suffix enables the update of the CR.
Floating Multiply (Double-Precision)	**fmul** **fmul.**	**fr**D,**fr**A,**fr**C	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. **fmul** Floating Multiply (Double-Precision) **fmul.** Floating Multiply (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Multiply Single	**fmuls** **fmuls.**	**fr**D,**fr**A,**fr**C	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. **fmuls** Floating Multiply Single **fmuls.** Floating Multiply Single with CR Update. The dot suffix enables the update of the CR.
Floating Divide (Double-Precision)	**fdiv** **fdiv.**	**fr**D,**fr**A,**fr**B	The floating-point operand in **fr**A is divided by the floating-point operand in **fr**B. No remainder is preserved. **fdiv** Floating Divide (Double-Precision) **fdiv.** Floating Divide (Double-Precision) with CR Update. The dot suffix enables the update of the CR.

Table 4-6. Floating-Point Arithmetic Instructions (continued)

Name	Mnemonic	Syntax	Operation
Floating Divide Single	**fdivs** **fdivs.**	**fr**D,**fr**A,**fr**B	The floating-point operand in **fr**A is divided by the floating-point operand in **fr**B. No remainder is preserved. **fdivs** Floating Divide Single **fdivs.** Floating Divide Single with CR Update. The dot suffix enables the update of the CR.
Floating Square Root (Double-Precision)	**fsqrt** **fsqrt.**	**fr**D,**fr**B	The square root of the floating-point operand in **fr**B is placed into **fr**D. **fsqrt** Floating Square Root (Double-Precision) **fsqrt.** Floating Square Root (Double-Precision) with CR Update. The dot suffix enables the update of the CR. This instruction is optional.
Floating Square Root Single	**fsqrts** **fsqrts.**	**fr**D,**fr**B	The square root of the floating-point operand in **fr**B is placed into **fr**D. **fsqrts** Floating Square Root Single **fsqrts.** Floating Square Root Single with CR Update. The dot suffix enables the update of the CR. This instruction is optional.
Floating Reciprocal Estimate Single	**fres** **fres.**	**fr**D,**fr**B	A single-precision estimate of the reciprocal of the floating-point operand in **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 256 of the reciprocal of **fr**B. **fres** Floating Reciprocal Estimate Single **fres.** Floating Reciprocal Estimate Single with CR Update. The dot suffix enables the update of the CR. This instruction is optional.
Floating Reciprocal Square Root Estimate	**frsqrte** **frsqrte.**	**fr**D,**fr**B	A double-precision estimate of the reciprocal of the square root of the floating-point operand in **fr**B is placed into **fr**D. The estimate placed into **fr**D is correct to a precision of one part in 32 of the reciprocal of the square root of **fr**B. **frsqrte** Floating Reciprocal Square Root Estimate **frsqrte.** Floating Reciprocal Square Root estimate with CR Update. The dot suffix enables the update of the CR. This instruction is optional.
Floating Select	**fsel**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is compared to the value zero. If the operand is greater than or equal to zero, **fr**D is set to the contents of **fr**C. If the operand is less than zero or is a *NaN*, **fr**D is set to the contents of **fr**B. The comparison ignores the sign of zero (that is, regards +0 as equal to –0). **fsel** Floating Select **fsel.** Floating Select with CR Update. The dot suffix enables the update of the CR. This instruction is optional.

4.2.2.2 Floating-Point Multiply-Add Instructions

Floating-point multiply-add instructions combine multiply and add operations without an intermediate rounding operation. The fractional part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows:

- Overflow, underflow, and inexact exception bits, FPSCR[FR,FI,FPRF] are set based on the final result of the operation and not on the result of the multiplication.
- Invalid operation exception bits are set as if the multiplication and the addition were performed using two separate instructions (**fmuls**, followed by **fadds** or **fsubs**). That is, multiplication of infinity by zero or of anything by an SNaN, and/or addition of an SNaN, cause the corresponding exception bits to be set.

The floating-point multiply-add instructions are summarized in Table 4-7.

Table 4-7. Floating-Point Multiply-Add Instructions

Name	Mnemonic	Syntax	Operation
Floating Multiply-Add (Double-Precision)	**fmadd** **fmadd.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result. **fmadd** Floating Multiply-Add (Double-Precision) **fmadd.** Floating Multiply-Add (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Multiply-Add Single	**fmadds** **fmadds.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result. **fmadds** Floating Multiply-Add Single **fmadds.** Floating Multiply-Add Single with CR Update. The dot suffix enables the update of the CR.
Floating Multiply-Subtract (Double-Precision)	**fmsub** **fmsub.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result. **fmsub** Floating Multiply-Subtract (Double-Precision) **fmsub.** Floating Multiply-Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Multiply-Subtract Single	**fmsubs** **fmsubs.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result. **fmsubs** Floating Multiply-Subtract Single **fmsubs.** Floating Multiply-Subtract Single with CR Update. The dot suffix enables the update of the CR.
Floating Negative Multiply-Add (Double-Precision)	**fnmadd** **fnmadd.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result. **fnmadd** Floating Negative Multiply-Add (Double-Precision) **fnmadd.** Floating Negative Multiply-Add (Double-Precision) with CR Update. The dot suffix enables update of the CR.
Floating Negative Multiply-Add Single	**fnmadds** **fnmadds.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is added to this intermediate result. **fnmadds** Floating Negative Multiply-Add Single **fnmadds.** Floating Negative Multiply-Add Single with CR Update. The dot suffix enables the update of the CR.

Table 4-7. Floating-Point Multiply-Add Instructions (continued)

Name	Mnemonic	Syntax	Operation
Floating Negative Multiply-Subtract (Double-Precision)	**fnmsub** **fnmsub.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result. **fnmsub** Floating Negative Multiply-Subtract (Double-Precision) **fnmsub.** Floating Negative Multiply-Subtract (Double-Precision) with CR Update. The dot suffix enables the update of the CR.
Floating Negative Multiply-Subtract Single	**fnmsubs** **fnmsubs.**	**fr**D,**fr**A,**fr**C,**fr**B	The floating-point operand in **fr**A is multiplied by the floating-point operand in **fr**C. The floating-point operand in **fr**B is subtracted from this intermediate result. **fnmsubs** Floating Negative Multiply-Subtract Single **fnmsubs.** Floating Negative Multiply-Subtract Single with CR Update. The dot suffix enables the update of the CR.

For more information on multiply-add instructions, refer to Section C.2, “Multiply-Add Type Instruction Execution Model.”

4.2.2.3 Floating-Point Rounding and Conversion Instructions

The Floating Round to Single-Precision (**frsp**) instruction is used to truncate a 64-bit double-precision number to a 32-bit single-precision floating-point number. The floating-point convert instructions convert a 64-bit double-precision floating-point number to a 32-bit signed integer number.

The architecture defines **fr**D[0–31] as undefined when executing the Floating Convert to Integer Word (**fctiw**) and Floating Convert to Integer Word with Round toward Zero (**fctiwz**) instructions. Floating-point rounding instructions are described in Table 4-8.

Examples of uses of these instructions to perform various conversions can be found in Appendix C, “Floating-Point Models.”

Table 4-8. Floating-Point Rounding and Conversion Instructions

Name	Mnemonic	Syntax	Operation
Floating Round to Single-Precision	**frsp** **frsp.**	**fr**D,**fr**B	The floating-point operand in **fr**B is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into **fr**D. **frsp** Floating Round to Single-Precision **frsp.** Floating Round to Single-Precision with CR Update. The dot suffix enables the update of the CR.
Floating Convert to Integer Word	**fctiw** **fctiw.**	**fr**D,**fr**B	The floating-point operand in **fr**B is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in the low-order 32 bits of **fr**D. Bits 0–31 of **fr**D are undefined. **fctiw** Floating Convert to Integer Word **fctiw.** Floating Convert to Integer Word with CR Update. The dot suffix enables the update of the CR.
Floating Convert to Integer Word with Round toward Zero	**fctiwz** **fctiwz.**	**fr**D,**fr**B	The floating-point operand in **fr**B is converted to a 32-bit signed integer, using the rounding mode Round toward Zero, and placed in the low-order 32 bits of **fr**D. Bits 0–31 of **fr**D are undefined. **fctiwz** Floating Convert to Integer Word with Round toward Zero **fctiwz.** Floating Convert to Integer Word with Round toward Zero with CR Update. The dot suffix enables the update of the CR.

4.2.2.4 Floating-Point Compare Instructions

Floating-point compare instructions compare the contents of two FPRs and the comparison ignores the sign of zero (that is +0 = –0). The comparison can be ordered or unordered. The comparison sets one bit in the designated CR field and clears the other three bits. The FPCC (floating-point condition code) FPSCD[16–19] is set in the same way.

The CR field and the FPCC are interpreted as shown in Table 4-9.

Table 4-9. CR Bit Settings

Bit	Name	Description
0	FL	(**fr**A) < (**fr**B)
1	FG	(**fr**A) > (**fr**B)
2	FE	(**fr**A) = (**fr**B)
3	FU	(**fr**A) ? (**fr**B) (unordered)

The floating-point compare instructions are summarized in Table 4-10.

Table 4-10. Floating-Point Compare Instructions

Name	Mnemonic	Syntax	Operation
Floating Compare Unordered	fcmpu	**crf**D,**fr**A,**fr**B	The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into **crf**D and the FPCC.
Floating Compare Ordered	fcmpo	**crf**D,**fr**A,**fr**B	The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into **crf**D and the FPCC.

4.2.2.5 Floating-Point Status and Control Register Instructions

Every FPSCR instruction appears to synchronize the effects of all floating-point instructions executed by a given processor. Executing an FPSCR instruction ensures that all floating-point instructions previously initiated by the given processor appear to have completed before the FPSCR instruction is initiated and that no subsequent floating-point instructions appear to be initiated by the given processor until the FPSCR instruction has completed. In particular:

- All exceptions caused by the previously initiated instructions are recorded in the FPSCR before the FPSCR instruction is initiated.
- All invocations of the floating-point exception handler caused by the previously initiated instructions have occurred before the FPSCR instruction is initiated.
- No subsequent floating-point instruction that depends on or alters the settings of any FPSCR bits appears to be initiated until the FPSCR instruction has completed.

Floating-point memory access instructions are not affected by the execution of the FPSCR instructions.

The FPSCR instructions are summarized in Table 4-11.

Table 4-11. Floating-Point Status and Control Register Instructions

Name	Mnemonic	Syntax	Operation
Move from FPSCR	**mffs** **mffs.**	**fr**D	The contents of the FPSCR are placed into bits 32–63 of **fr**D. **fr**D[0–31] are undefined. **mffs** Move from FPSCR **mffs.** Move from FPSCR with CR Update. The dot suffix enables the update of the CR.
Move to Condition Register from FPSCR	**mcrfs**	**crf**D,**crf**S	The contents of FPSCR field specified by operand **crf**S are copied to the CR field specified by operand **crf**D. All exception bits copied (except FEX and VX) are cleared in the FPSCR.
Move to FPSCR Field Immediate	**mtfsfi** **mtfsfi.**	**crf**D,IMM	The contents of the IMM field are placed into FPSCR field **crf**D. The contents of FPSCR[FX] are altered only if **crf**D = 0. **mtfsfi** Move to FPSCR Field Immediate **mtfsfi.** Move to FPSCR Field Immediate with CR Update. The dot suffix enables the update of the CR.
Move to FPSCR Fields	**mtfsf** **mtfsf.**	FM,**fr**B	Bits 32–63 of **fr**B are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0–7. If FM[*i*] = 1, FPSCR field *i* (FPSCR bits 4**i* through 4**i*+3) is set to the contents of the corresponding field of the low-order 32 bits of **fr**B. The contents of FPSCR[FX] are altered only if FM[0] = 1. **mtfsf** Move to FPSCR Fields **mtfsf.** Move to FPSCR Fields with CR Update. The dot suffix enables the update of the CR.
Move to FPSCR Bit 0	**mtfsb0** **mtfsb0.**	**crb**D	The FPSCR bit location specified by operand **crb**D is cleared. Bits 1 and 2 (FEX and VX) cannot be reset explicitly. **mtfsb0** Move to FPSCR Bit 0 **mtfsb0.** Move to FPSCR Bit 0 with CR Update. The dot suffix enables the update of the CR.
Move to FPSCR Bit 1	**mtfsb1** **mtfsb1.**	**crb**D	The FPSCR bit location specified by operand **crb**D is set. Bits 1 and 2 (FEX and VX) cannot be set explicitly. **mtfsb1** Move to FPSCR Bit 1 **mtfsb1.** Move to FPSCR Bit 1 with CR Update. The dot suffix enables the update of the CR.

4.2.2.6 Floating-Point Move Instructions

Floating-point move instructions copy data from one FPR to another, altering the sign bit (bit 0) as described for the **fneg**, **fabs**, and **fnabs** instructions in Table 4-12; **fneg**, **fabs**, and **fnabs** may alter the sign bit of a NaN. Floating-point move instructions do not modify the FPSCR. The CR update option in these instructions controls the placing of result status into CR1. If the CR update option is enabled, CR1 is set; otherwise, CR1 is unchanged.

Table 4-12 provides a summary of the floating-point move instructions.

Table 4-12. Floating-Point Move Instructions

Name	Mnemonic	Syntax	Operation
Floating Move Register	**fmr** **fmr.**	**fr**D,**fr**B	The contents of **fr**B are placed into **fr**D. **fmr** Floating Move Register **fmr.** Floating Move Register with CR Update. The dot suffix enables the update of the CR.
Floating Negate	**fneg** **fneg.**	**fr**D,**fr**B	The contents of **fr**B with bit 0 inverted are placed into **fr**D. **fneg** Floating Negate **fneg.** Floating Negate with CR Update. The dot suffix enables the update of the CR.
Floating Absolute Value	**fabs** **fabs.**	**fr**D,**fr**B	The contents of **fr**B with bit 0 cleared are placed into **fr**D. **fabs** Floating Absolute Value **fabs.** Floating Absolute Value with CR Update. The dot suffix enables the update of the CR.
Floating Negative Absolute Value	**fnabs** **fnabs.**	**fr**D,**fr**B	The contents of **fr**B with bit 0 set are placed into **fr**D. **fnabs** Floating Negative Absolute Value **fnabs.** Floating Negative Absolute Value with CR Update. The dot suffix enables the update of the CR.

4.2.3 Load and Store Instructions

Load and store instructions are issued and translated in program order; however, the accesses can occur out of order. Synchronizing instructions are provided to enforce strict ordering. This section describes the load and store instructions, which consist of the following:

- Integer load instructions
- Integer store instructions
- Integer load and store with byte-reverse instructions
- Integer load and store multiple instructions
- Floating-point load instructions
- Floating-point store instructions
- Memory synchronization instructions

4.2.3.1 Integer Load and Store Address Generation

Integer load and store operations generate effective addresses using register indirect with immediate index mode, register indirect with index mode, or register indirect mode. See Section 4.1.3.2, “Effective Address Calculation,” for information about calculating effective addresses. Note that in some implementations, operations that are not naturally aligned may suffer performance degradation. Section 6.5.6.1, “Integer Alignment Interrupts,” gives additional information about load and store address alignment interrupts.

4.2.3.1.1 Register Indirect with Immediate Index Addressing for Integer Loads and Stores

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended, and added to the contents of a general-purpose register specified in the instruction (**r**A operand) to generate the effective address. If the **r**A field specifies **r0**, a value of zero is added to the immediate index (d operand) in place of the contents of **r0**. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 4-1 shows how an effective address is generated when using register indirect with immediate index addressing.

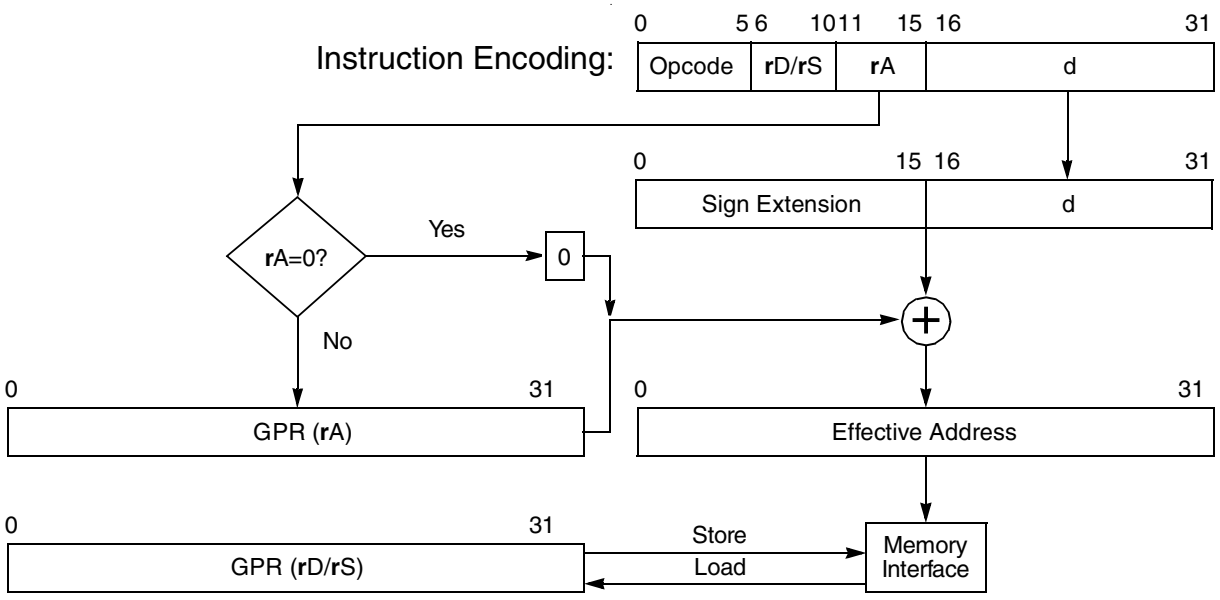


Figure 4-1. Register Indirect with Immediate Index Addressing for Integer Loads/Stores

4.2.3.1.2 Register Indirect with Index Addressing for Integer Loads and Stores

Instructions using this addressing mode cause the contents of two GPRs (specified as operands **r**A and **r**B) to be added in the generation of the effective address. A zero in place of the **r**A operand causes a zero to be added to the contents of the general-purpose register specified in operand **r**B (or the value zero for **lswi** and **stswi** instructions). The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 4-2 shows how an effective address is generated when using register indirect with index addressing.

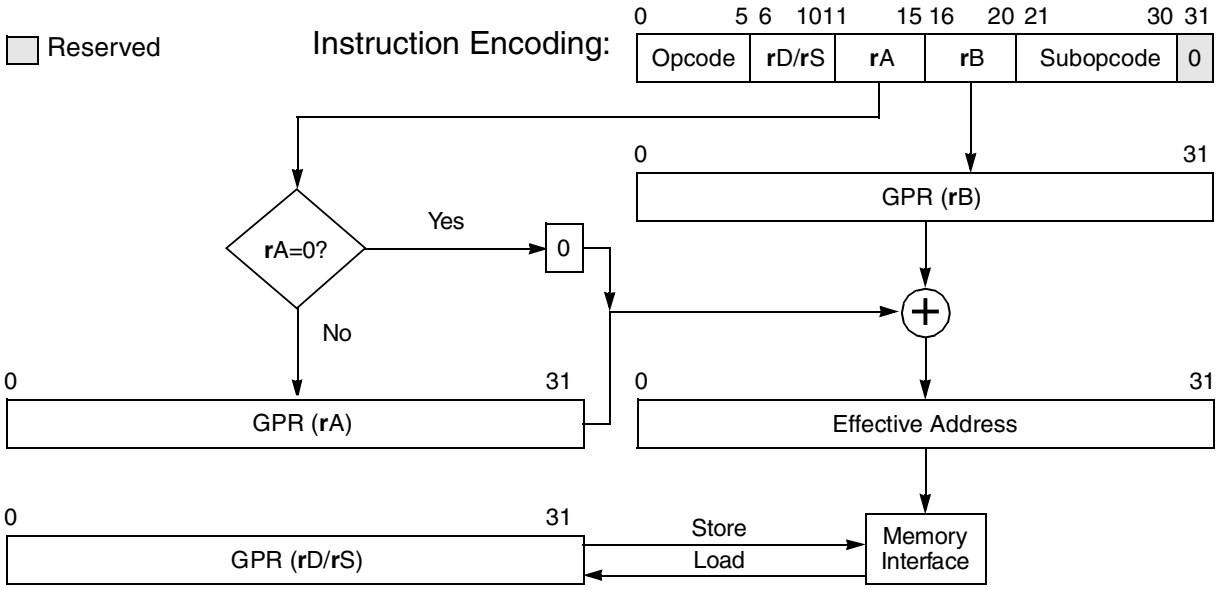


Figure 4-2. Register Indirect with Index Addressing for Integer Loads/Stores

4.2.3.1.3 Register Indirect Addressing for Integer Loads and Stores

Instructions using this addressing mode use the contents of the GPR specified by the **r**A operand as the effective address. A zero in the **r**A operand causes an effective address of zero to be generated. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 4-3 shows how an effective address is generated when using register indirect addressing.

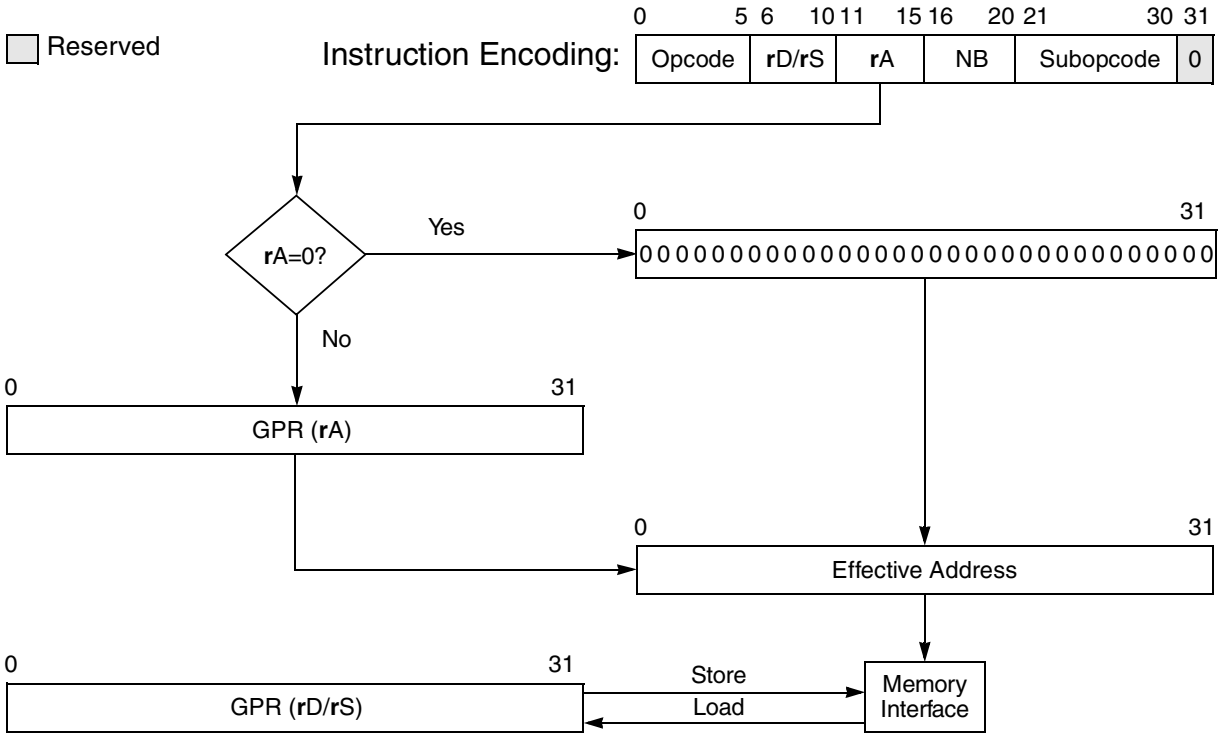


Figure 4-3. Register Indirect Addressing for Integer Loads/Stores

4.2.3.2 Integer Load Instructions

For integer load instructions, the byte, half word, word, or double word addressed by the EA (effective address) is loaded into **r**D. Many integer load instructions have an update form, in which **r**A is updated with the generated effective address. For these forms, if **r**A ≠ 0 and **r**A ≠ **r**D (otherwise invalid), the EA is placed into **r**A and the memory element (byte, half word, word, or double word) addressed by the EA is loaded into **r**D. Note that the architecture defines load with update instructions with operand **r**A = 0 or **r**A = **r**D as invalid forms.

The default byte and bit ordering is big-endian in the architecture; see Section 3.1.2, “Byte Ordering.”

Note that in some implementations of the architecture, the load word algebraic instructions (**lha**, **lhax**, **lwa**, **lwax**) and the load with update (**lbzu**, **lbzux**, **lhzu**, **lhzux**, **lhau**, **lhaux**, **lwaux**, **ldu**, **ldux**) instructions may execute with greater latency than other types of load instructions. Moreover, the load with update instructions may take longer to execute in some implementations than the corresponding pair of a nonupdate load followed by an add instruction.

Table 4-13 summarizes the integer load instructions.

Table 4-13. Integer Load Instructions

Name	Mnemonic	Syntax	Operation
Load Byte and Zero	**lbz**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The byte in memory addressed by the EA is loaded into the low-order 8 bits of **r**D. The remaining bits in **r**D are cleared.
Load Byte and Zero Indexed	**lbzx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The byte in memory addressed by the EA is loaded into the low-order 8 bits of **r**D. The remaining bits in **r**D are cleared.
Load Byte and Zero with Update	**lbzu**	**r**D,d(**r**A)	The EA is the sum (**r**A) + d. The byte in memory addressed by the EA is loaded into the low-order 8 bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Byte and Zero with Update Indexed	**lbzux**	**r**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The byte in memory addressed by the EA is loaded into the low-order 8 bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Half Word and Zero	**lhz**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.
Load Half Word and Zero Indexed	**lhzx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.
Load Half Word and Zero with Update	**lhzu**	**r**D,d(**r**A)	The EA is the sum (**r**A) + d. The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Half Word and Zero with Update Indexed	**lhzux**	**r**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared. The EA is placed into **r**A.
Load Half Word Algebraic	**lha**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded half word.
Load Half Word Algebraic Indexed	**lhax**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded half word.
Load Half Word Algebraic with Update	**lhau**	**r**D,d(**r**A)	The EA is the sum (**r**A) + d. The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded half word. The EA is placed into **r**A.
Load Half Word Algebraic with Update Indexed	**lhaux**	**r**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The half word in memory addressed by the EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the most significant bit of the loaded half word. The EA is placed into **r**A.
Load Word and Zero	**lwz**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The word in memory addressed by the EA is loaded into **r**D.
Load Word and Zero Indexed	**lwzx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into **r**D.

Table 4-13. Integer Load Instructions (continued)

Name	Mnemonic	Syntax	Operation
Load Word and Zero with Update	**lwzu**	**r**D,d(**r**A)	The EA is the sum (**r**A) + d. The word in memory addressed by the EA is loaded into **r**D. The EA is placed into **r**A.
Load Word and Zero with Update Indexed	**lwzux**	**r**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The word in memory addressed by the EA is loaded into **r**D. The EA is placed into **r**A.

4.2.3.3 Integer Store Instructions

For integer store instructions, the contents of **r**S are stored into the byte, half word, word or double word in memory addressed by the EA (effective address). Many store instructions have an update form, in which **r**A is updated with the EA. For these forms, the following rules apply:

- If **r**A ≠ 0, the effective address is placed into **r**A.
- If **r**S = **r**A, the contents of register **r**S are copied to the target memory element, then the generated EA is placed into **r**A (**r**S).

In general, the architecture defines a sequential execution model. However, when a store instruction modifies a location that contains an instruction, software synchronization is required to ensure that subsequent instruction fetches from that location obtain the modified version of the instruction.

If a program modifies the instructions it intends to execute, it should call the appropriate system library program before attempting to execute the modified instructions to ensure that the modifications have taken effect with respect to instruction fetching.

The architecture defines store with update instructions with **r**A = 0 as an invalid form. In addition, it defines integer store instructions with the CR update option enabled (Rc field, bit 31, in the instruction encoding = 1) to be an invalid form. Table 4-14 provides a summary of the integer store instructions.

Table 4-14. Integer Store Instructions

Name	Mnemonic	Syntax	Operation
Store Byte	**stb**	**r**S,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of the low-order 8 bits of **r**S are stored into the byte in memory addressed by the EA.
Store Byte Indexed	**stbx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 8 bits of **r**S are stored into the byte in memory addressed by the EA.
Store Byte with Update	**stbu**	**r**S,d(**r**A)	The EA is the sum (**r**A) + d. The contents of the low-order 8 bits of **r**S are stored into the byte in memory addressed by the EA. The EA is placed into **r**A.
Store Byte with Update Indexed	**stbux**	**r**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of the low-order 8 bits of **r**S are stored into the byte in memory addressed by the EA. The EA is placed into **r**A.
Store Half Word	**sth**	**r**S,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by the EA.
Store Half Word Indexed	**sthx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by the EA.

Table 4-14. Integer Store Instructions (continued)

Name	Mnemonic	Syntax	Operation
Store Half Word with Update	**sthu**	**r**S,d(**r**A)	The EA is the sum (**r**A) + d. The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by the EA. The EA is placed into **r**A.
Store Half Word with Update Indexed	**sthux**	**r**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by the EA. The EA is placed into **r**A.
Store Word	**stw**	**r**S,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of **r**S are stored into the word in memory addressed by the EA.
Store Word Indexed	**stwx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of **r**S are stored into the word in memory addressed by the EA.
Store Word with Update	**stwu**	**r**S,d(**r**A)	The EA is the sum (**r**A) + d. The contents of **r**S are stored into the word in memory addressed by the EA. The EA is placed into **r**A.
Store Word with Update Indexed	**stwux**	**r**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of **r**S are stored into the word in memory addressed by the EA. The EA is placed into **r**A.

4.2.3.4 Integer Load and Store with Byte-Reverse Instructions

Table 4-15 describes integer load and store with byte-reverse instructions. Note that in some implementations, load byte-reverse instructions may have greater latency than other load instructions.

When used in a system operating with the default big-endian byte order, these instructions have the effect of loading and storing data in little-endian order. Likewise, when used in a system operating with little-endian byte order, these instructions have the effect of loading and storing data in big-endian order. For more information about big-endian and little-endian byte ordering, see Section 3.1.2, “Byte Ordering.”

Table 4-15. Integer Load and Store with Byte-Reverse Instructions

Name	Mnemonic	Syntax	Operation
Load Half Word Byte-Reverse Indexed	**lhbrx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The high-order 8 bits of the half word addressed by the EA are loaded into the low-order 8 bits of **r**D. The next eight higher-order bits of the half word in memory addressed by the EA are loaded into the next eight lower-order bits of **r**D. The remaining **r**D bits are cleared.
Load Word Byte-Reverse Indexed	**lwbrx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). Bits 0–7 of the word in memory addressed by the EA are loaded into the low-order 8 bits of **r**D. Bits 8–15 of the word in memory addressed by the EA are loaded into bits 16–23 of **r**D. Bits 16–23 of the word in memory addressed by the EA are loaded into bits 8–15. Bits 24–31 of the word in memory addressed by the EA are loaded into bits 0–7. The remaining bits in **r**D are cleared.

Table 4-15. Integer Load and Store with Byte-Reverse Instructions (continued)

Name	Mnemonic	Syntax	Operation
Store Half Word Byte-Reverse Indexed	sthbrx	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 8 bits of **r**S are stored into the high-order 8 bits of the half word in memory addressed by the EA. The contents of the next lower-order 8 bits of **r**S are stored into the next eight higher-order bits of the half word in memory addressed by the EA.
Store Word Byte-Reverse Indexed	stwbrx	**r**S,**r**A,**r**B	The effective address is the sum (**r**A|0) + (**r**B). The contents of the low-order 8 bits of **r**S are stored into bits 0–7 of the word in memory addressed by EA. The contents of the next eight lower-order bits of **r**S are stored into bits 8–15 of the word in memory addressed by the EA. The contents of the next eight lower-order bits of **r**S are stored into bits 16–23 of the word in memory addressed by the EA. The contents of the next eight lower-order bits of **r**S are stored into bits 24–31 of the word addressed by the EA.

4.2.3.5 Integer Load and Store Multiple Instructions

The load/store multiple instructions are used to move blocks of data to and from the GPRs. The load multiple and store multiple instructions may have operands that require memory accesses crossing a 4-Kbyte page boundary. As a result, these instructions may be interrupted by a DSI interrupt associated with the address translation of the second page. Table 4-16 summarizes the integer load and store multiple instructions.

In the load/store multiple instructions, the combination of the EA and **r**D (**r**S) is such that the low-order byte of GPR31 is loaded from or stored into the last byte of an aligned quad word in memory; if the effective address is not correctly aligned, it may take significantly longer to execute.

In some implementations operating with little-endian byte order, execution of an **lmw** or **stmw** instruction causes the system alignment error handler to be invoked; see Section 3.1.2, “Byte Ordering.”

The architecture defines the load multiple word (**lmw**) instruction with **r**A in the range of registers to be loaded, including the case in which **r**A = 0, as an invalid form.

Table 4-16. Integer Load and Store Multiple Instructions

Name	Mnemonic	Syntax	Operation
Load Multiple Word	**lmw**	**r**D,d(**r**A)	The EA is the sum (**r**A|0) + d. *n* = (32 – **r**D).
Store Multiple Word	**stmw**	**r**S,d(**r**A)	The EA is the sum (**r**A|0) + d. *n* = (32 – **r**S).

4.2.3.6 Integer Load and Store String Instructions

The integer load and store string instructions allow movement of data from memory to registers or from registers to memory without concern for alignment. These instructions can be used for a short move between arbitrary memory locations or to initiate a long move between misaligned memory fields. However, in some implementations, these instructions are likely to have greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results. Table 4-17 summarizes the integer load and store string instructions.

Load and store string instructions execute more efficiently when **r**D or **r**S = 5, and the last register loaded or stored is less than or equal to 12.

In some implementations operating with little-endian byte order, execution of a load or string instruction causes the system alignment error handler to be invoked; see Section 3.1.2, “Byte Ordering,” for more information.

Table 4-17. Integer Load and Store String Instructions

Name	Mnemonic	Syntax	Operation
Load String Word Immediate	**lswi**	**r**D,**r**A,NB	The EA is (**r**A|0).
Load String Word Indexed	**lswx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B).
Store String Word Immediate	**stswi**	**r**S,**r**A,NB	The EA is (**r**A|0).
Store String Word Indexed	**stswx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B).

Load string and store string instructions may involve operands that are not word-aligned. As described in Section 6.5.6, “Alignment Interrupt (0x00600),” a misaligned string operation suffers a performance penalty compared to an aligned operation of the same type. A non–word-aligned string operation that crosses a double-word boundary is also slower than a word-aligned string operation.

4.2.3.7 Floating-Point Load and Store Address Generation

Floating-point load and store operations generate effective addresses using the register indirect with immediate index addressing mode and register indirect with index addressing mode.

4.2.3.7.1 Register Indirect with Immediate Index Addressing for Floating-Point Loads and Stores

Instructions using this addressing mode contain a signed 16-bit immediate index (d operand) which is sign extended to 32 bits, and added to the contents of a GPR specified in the instruction (**r**A operand) to generate the effective address. If the **r**A field of the instruction specifies **r0**, a value of zero is added to the immediate index (d operand) in place of the contents of **r0**. The option to specify **r**A or 0 is shown in the instruction descriptions as (**r**A|0).

Figure 4-4 shows how an effective address is generated when using register indirect with immediate index addressing for floating-point loads and stores.

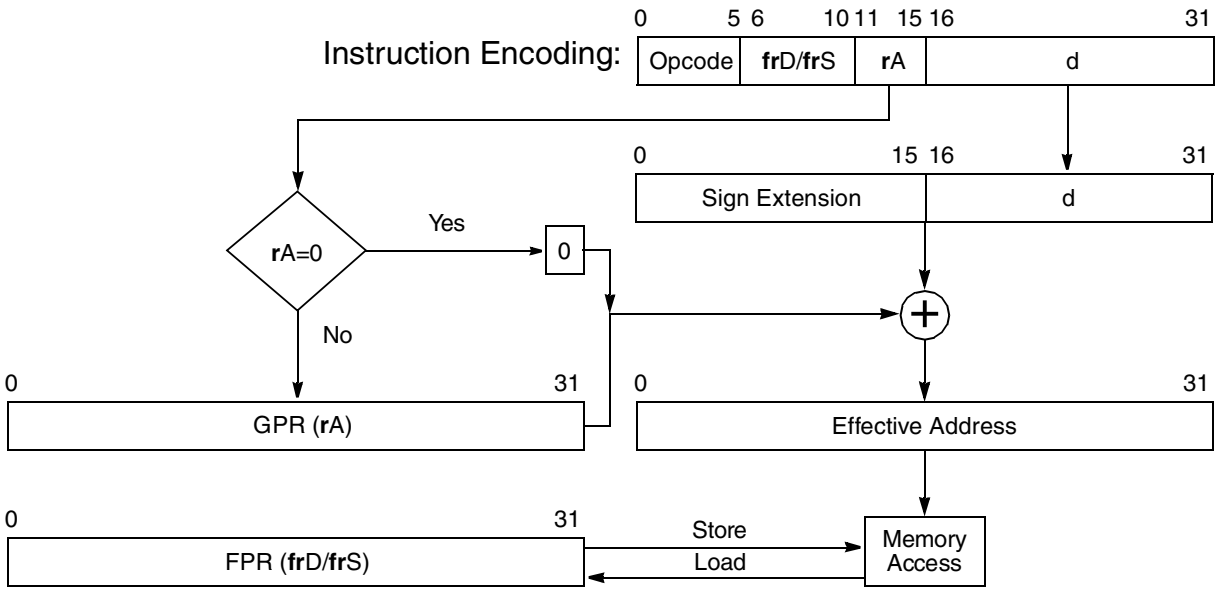


Figure 4-4. Register Indirect with Immediate Index Addressing for Floating-Point Loads/Stores

4.2.3.7.2 Register Indirect with Index Addressing for Floating-Point Loads and Stores

Instructions using this addressing mode add the contents of two GPRs (specified in operands **r**A and **r**B) to generate the effective address. A zero in the **r**A operand causes a zero to be added to the contents of the GPR specified in operand **r**B. This is shown in the instruction descriptions as (**r**A|0).

Figure 4-5 shows how an effective address is generated for register indirect with index addressing.

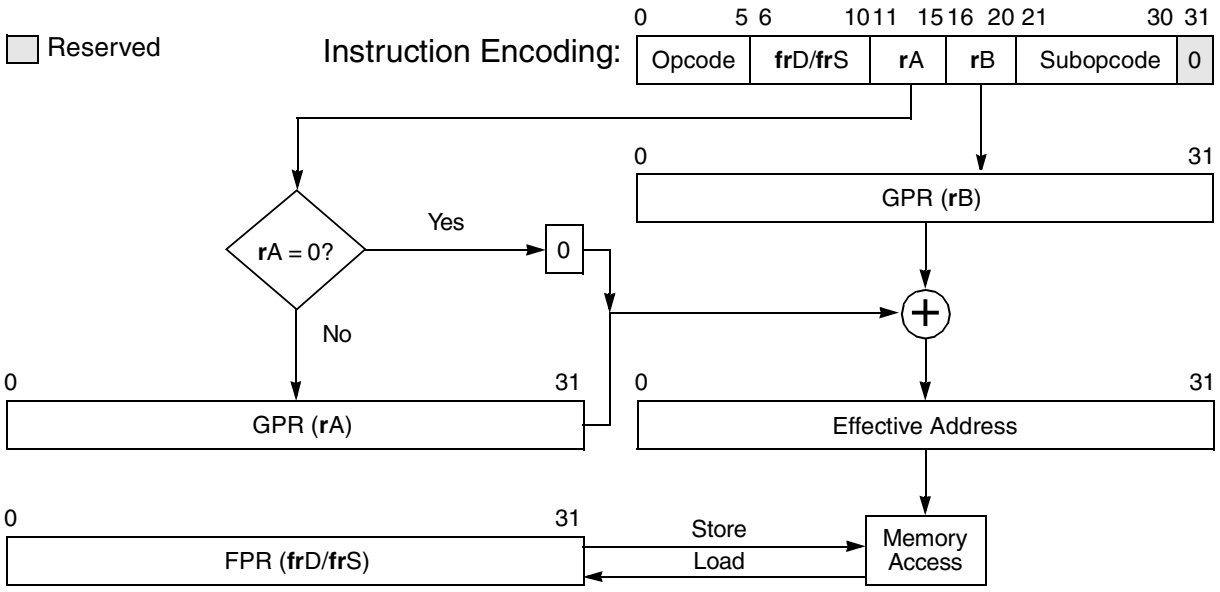


Figure 4-5. Register Indirect with Index Addressing for Floating-Point Loads/Stores

The architecture defines floating-point load and store with update instructions (**lfsu**, **lfsux**, **lfdu**, **lfdux**, **stfsu**, **stfsux**, **stfdu**, **stfdux**) with operand **r**A = 0 as invalid forms of the instructions. In addition, it defines floating-point load and store instructions with the CR updating option enabled (Rc bit, bit 31 = 1) to be an invalid form.

The architecture defines that FPSCR[UE] should not be used to determine whether denormalization should be performed on floating-point stores.

4.2.3.8 Floating-Point Load Instructions

There are two forms of the floating-point load instruction—single-precision and double-precision operand formats. Because the FPRs support only the floating-point double-precision format, single-precision floating-point load instructions convert single-precision data to double-precision format before loading the operands into the target FPR. This conversion is described fully in Section C.6, “Floating-Point Load Instructions.” Table 4-18 provides a summary of the floating-point load instructions.

Note that the architecture defines load with update instructions with **r**A = 0 as an invalid form.

Table 4-18. Floating-Point Load Instructions

Name	Mnemonic	Syntax	Operation
Load Floating-Point Single	**lfs**	**fr**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D.
Load Floating-Point Single Indexed	**lfsx**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D.
Load Floating-Point Single with Update	**lfsu**	**fr**D,d(**r**A)	The EA is the sum (**r**A) + d. The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Single with Update Indexed	**lfsux**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The word in memory addressed by the EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision format and placed into **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Double	**lfd**	**fr**D,d(**r**A)	The EA is the sum (**r**A|0) + d. The double word in memory addressed by the EA is placed into **fr**D.
Load Floating-Point Double Indexed	**lfdx**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The double word in memory addressed by the EA is placed into **fr**D.
Load Floating-Point Double with Update	**lfdu**	**fr**D,d(**r**A)	The EA is the sum (**r**A) + d. The double word in memory addressed by the EA is placed into **fr**D. The EA is placed into the register specified by **r**A.
Load Floating-Point Double with Update Indexed	**lfdux**	**fr**D,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The double word in memory addressed by the EA is placed into **fr**D. The EA is placed into the register specified by **r**A.

4.2.3.9 Floating-Point Store Instructions

This section describes floating-point store instructions. There are three basic forms of the store instruction—single-precision, double-precision, and integer. The integer form is supported by the **stfiwx** instruction. (Note that the **stfiwx** instruction is defined as optional by the architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent processors.) Because the FPRs support only floating-point, double-precision format for floating-point data, single-precision floating-point store instructions convert double-precision data to single-precision format before storing the operands. The conversion steps are described fully in Section C.7, “Floating-Point Store Instructions.” Table 4-19 provides a summary of the floating-point store instructions.

Note that the architecture defines store with update instructions with **r**A = 0 as an invalid form.

Table 4-19 provides the floating-point store instructions.

Table 4-19. Floating-Point Store Instructions

Name	Mnemonic	Syntax	Operation
Store Floating-Point Single	**stfs**	**fr**S,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single Indexed	**stfsx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA.
Store Floating-Point Single with Update	**stfsu**	**fr**S,d(**r**A)	The EA is the sum (**r**A) + d. The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into **r**A.
Store Floating-Point Single with Update Indexed	**stfsux**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by the EA. The EA is placed into the **r**A.
Store Floating-Point Double	**stfd**	**fr**S,d(**r**A)	The EA is the sum (**r**A|0) + d. The contents of **fr**S are stored into the double word in memory addressed by the EA.
Store Floating-Point Double Indexed	**stfdx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of **fr**S are stored into the double word in memory addressed by the EA.
Store Floating-Point Double with Update	**stfdu**	**fr**S,d(**r**A)	The EA is the sum (**r**A) + d. The contents of **fr**S are stored into the double word in memory addressed by the EA. The EA is placed into **r**A.
Store Floating-Point Double with Update Indexed	**stfdux**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A) + (**r**B). The contents of **fr**S are stored into the double word in memory addressed by EA. The EA is placed into register **r**A.
Store Floating-Point as Integer Word Indexed	**stfiwx**	**fr**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 32 bits of **fr**S are stored, without conversion, into the word in memory addressed by the EA. **Note**: **stfiwx** is defined as optional by the architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent processors.

4.2.4 Branch and Flow Control Instructions

Some branch instructions can redirect instruction execution conditionally based on the value of bits in the CR. When the processor encounters one of these instructions, it scans the execution pipelines to determine whether an instruction in progress may affect the particular CR bit. If no interlock is found, the branch can

be resolved immediately by checking the bit in the CR and taking the action defined for the branch instruction.

If an interlock is detected, the branch is considered unresolved and the direction of the branch may either be predicted using the *y* bit (as described in Table 4-20) or by using dynamic prediction. The interlock is monitored while instructions are fetched for the predicted branch. When the interlock is cleared, the processor determines whether the prediction was correct based on the value of the CR bit. If the prediction is correct, the branch is considered completed and instruction fetching continues. If the prediction is incorrect, the fetched instructions are purged, and instruction fetching continues along the alternate path.

4.2.4.1 Branch Instruction Address Calculation

Branch instructions can alter the sequence of instruction execution. Instruction addresses are always assumed to be word aligned; the two low-order bits of the generated branch target address are ignored.

Branch instructions compute the effective address (EA) of the next instruction address using the following addressing modes:

- Branch relative
- Branch conditional to relative address
- Branch to absolute address
- Branch conditional to absolute address
- Branch conditional to link register
- Branch conditional to count register

4.2.4.1.1 Branch Relative Addressing Mode

Instructions that use branch relative addressing generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand LI, and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The link register (LR) update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-6 shows how the branch target address is generated when using the branch relative addressing mode.

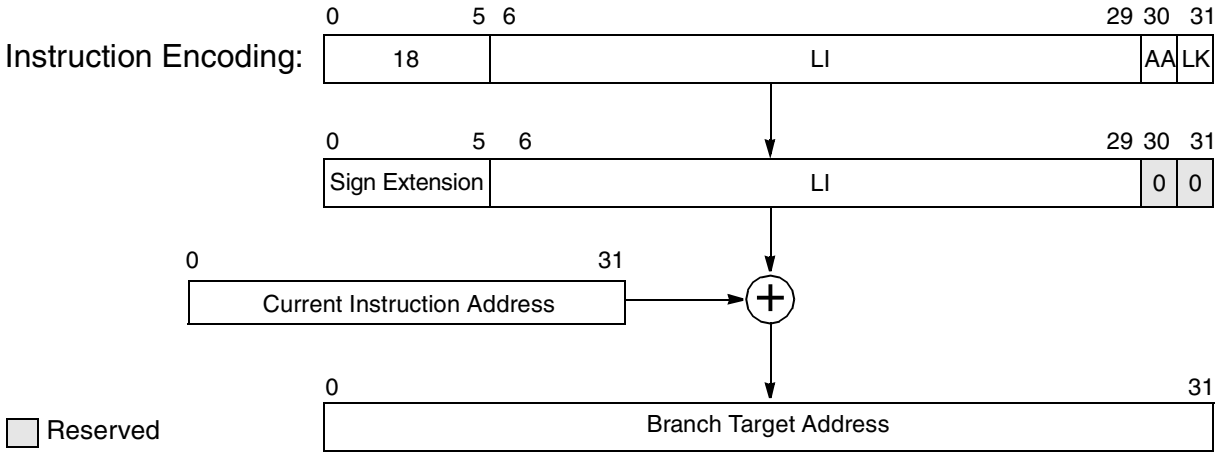


Figure 4-6. Branch Relative Addressing

4.2.4.1.2 Branch Conditional to Relative Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to relative addressing mode generate the next instruction address by sign extending and appending 0b00 to the immediate displacement operand (BD) and adding the resultant value to the current instruction address. Branches using this addressing mode have the absolute addressing option disabled (AA field, bit 30, in the instruction encoding = 0). The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-7 shows how the branch target address is generated when using the branch conditional relative addressing mode.

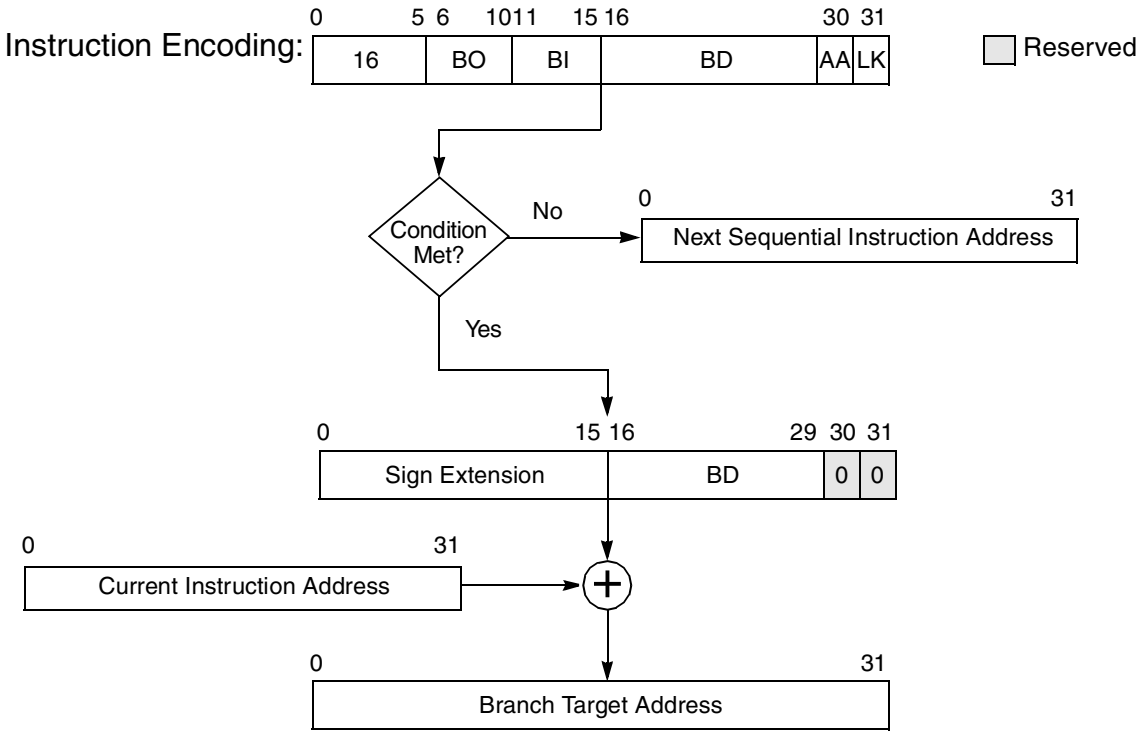


Figure 4-7. Branch Conditional Relative Addressing

4.2.4.1.3 Branch to Absolute Addressing Mode

Instructions that use branch to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the LI operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-8 shows how the branch target address is generated when using the branch to absolute addressing mode.

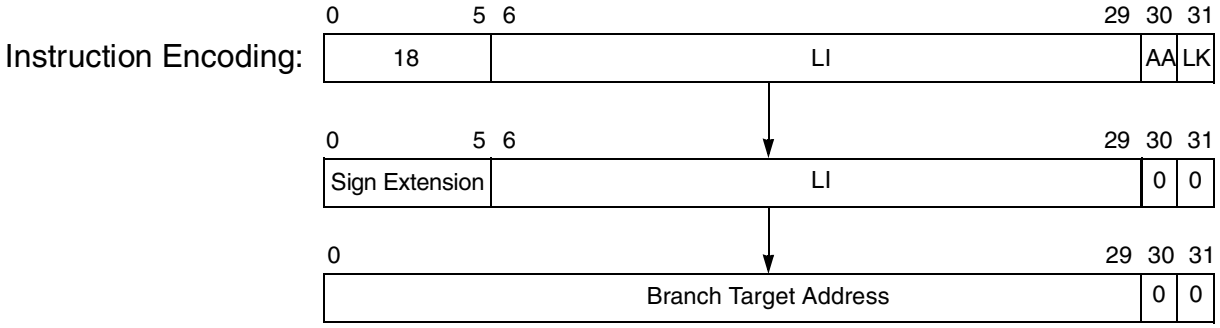


Figure 4-8. Branch to Absolute Addressing

4.2.4.1.4 Branch Conditional to Absolute Addressing Mode

If the branch conditions are met, instructions that use the branch conditional to absolute addressing mode generate the next instruction address by sign extending and appending 0b00 to the BD operand. Branches using this addressing mode have the absolute addressing option enabled (AA field, bit 30, in the instruction encoding = 1). The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-9 shows how the branch target address is generated when using the branch conditional to absolute addressing mode.

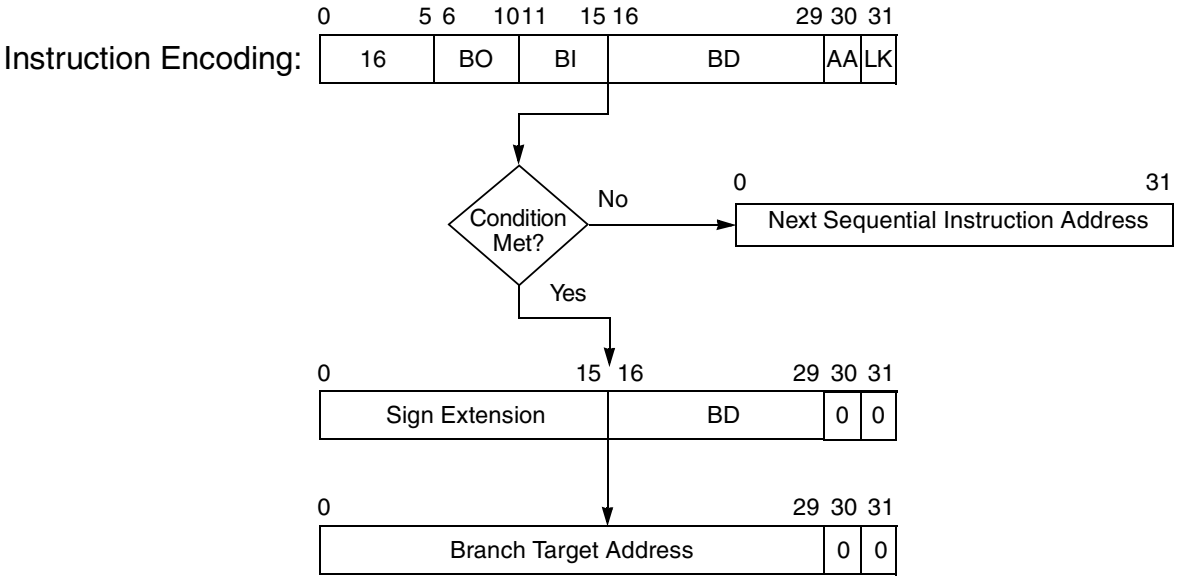


Figure 4-9. Branch Conditional to Absolute Addressing

4.2.4.1.5 Branch Conditional to Link Register Addressing Mode

If the branch conditions are met, the branch conditional to link register instruction generates the next instruction address by fetching the contents of the LR and clearing the two low-order bits to zero. The LR update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-10 shows how the branch target address is generated when using the branch conditional to link register addressing mode.

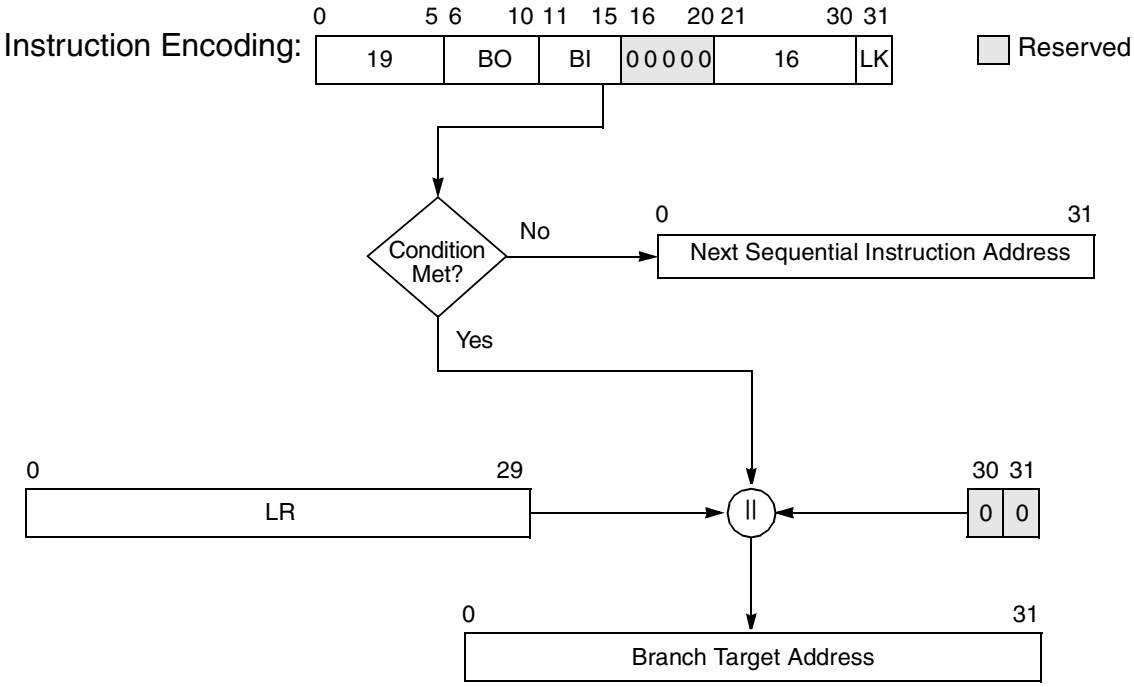


Figure 4-10. Branch Conditional to Link Register Addressing

4.2.4.1.6 Branch Conditional to Count Register Addressing Mode

If the branch conditions are met, the branch conditional to count register instruction generates the next instruction address by fetching the contents of the count register (CTR) and clearing the two low-order bits to zero. The link register update option can be enabled (LK field, bit 31, in the instruction encoding = 1). This option causes the effective address of the instruction following the branch instruction to be placed in the LR.

Figure 4-11 shows how the branch target address is generated when using the branch conditional to count register addressing mode.

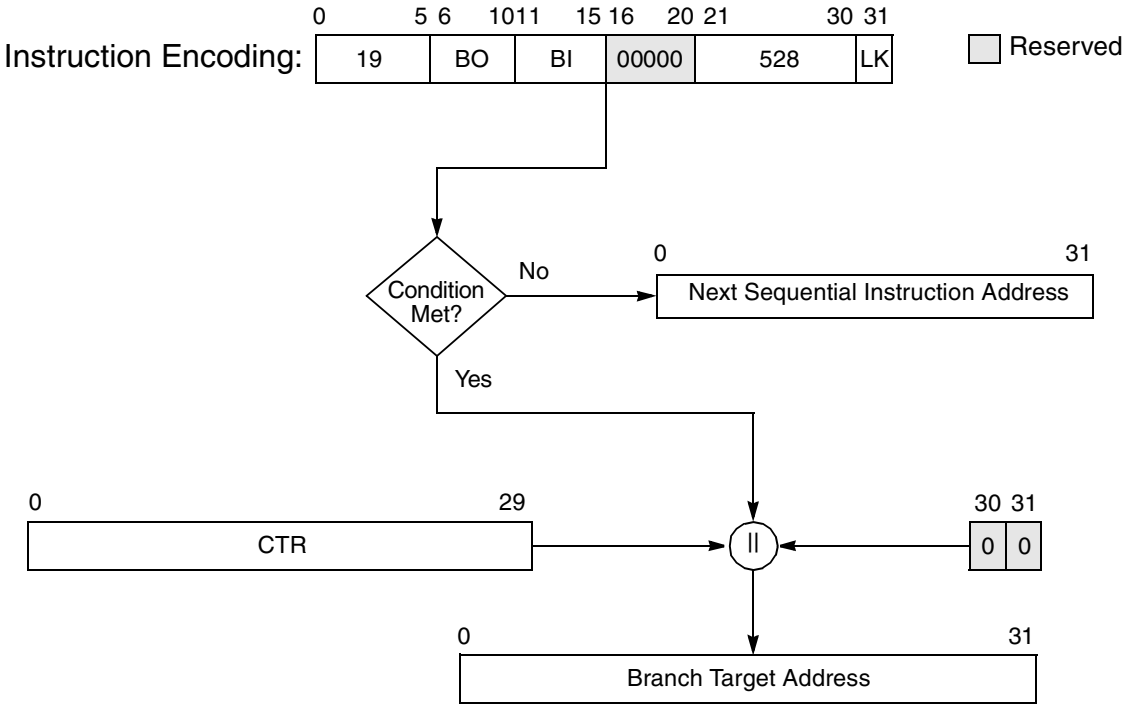


Figure 4-11. Branch Conditional to Count Register Addressing

4.2.4.2 Conditional Branch Control

For branch conditional instructions, the BO operand specifies the conditions under which the branch is taken. The first four bits of the BO operand specify how the branch is affected by or affects the CR and CTR. The fifth bit, shown in Table 4-20 as having the value *y*, is used by some implementations for branch prediction as described as part of the BO operand encodings in Table 4-20.

Table 4-20. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.

Table 4-20. BO Operand Encodings (continued)

BO	Description
1*z*1*zz*	Branch always

In this table, *z* indicates a bit that is ignored.
Note that the *z* bits should be cleared, as they may be assigned a meaning in some future version of the architecture.

The *y* bit provides a hint about whether a conditional branch is likely to be taken, and may be used by some implementations to improve performance.

The branch always encoding of the BO operand does not have a *y* bit.

Clearing the *y* bit indicates a predicted behavior for the branch instruction as follows:

- For **bc***x* with a negative value in the displacement operand, the branch is taken.
- In all other cases (**bc***x* with a non-negative value in the displacement operand, **bclr***x*, or **bcctr***x*), the branch is not taken.

Setting the *y* bit reverses the preceding indications.

The sign of the displacement operand is used as described above even if the target is an absolute address. The default value for the *y* bit should be 0, and should only be set to 1 if software has determined that the prediction corresponding to $y = 1$ is more likely to be correct than the prediction corresponding to $y = 0$. Software that does not compute branch predictions should clear the *y* bit.

In most cases, the branch should be predicted to be taken if the value of the following expression is 1, and predicted to fall through if the value is 0.

((BO[0] & BO[2]) | S) ≈ BO[4]

In the expression above, S (bit 16 of the branch conditional instruction coding) is the sign bit of the displacement operand if the instruction has a displacement operand and is 0 if the operand is reserved. BO[4] is the *y* bit, or 0 for the branch always encoding of the BO operand. (Advantage is taken of the fact that, for **bclr***x* and **bcctr***x*, bit 16 of the instruction is part of a reserved operand and therefore must be 0.)

The 5-bit BI operand in branch conditional instructions specifies which of the 32 CR bits represents the condition to test.

If a branch instruction contains an immediate addressing operand, the target addresses can be computed sufficiently ahead of the branch instruction that instructions can be fetched along the target path. If a branch instruction uses LR or CTR, instructions along the target path can be fetched if the LR or CTR is loaded sufficiently ahead of the branch instruction.

Branching can be conditional or unconditional, and optionally a branch return address is created by the access of the effective address of the instruction following the branch instruction in the LR after the branch target address has been computed. This is done regardless of whether the branch is taken. Some processors may keep a stack of the link register values most recently set by branch and link instructions, with the possible exception of the form shown below for obtaining the address of the next instruction. To benefit from this stack, the following programming conventions should be used.

In the following examples, let A, B, and Glue represent subroutine labels:

- Obtaining the address of the next instruction—use the following form of branch and link:

  bcl 20,31,$+4

- Loop counts:

  Keep them in the CTR, and use one of the branch conditional instructions to decrement the count and to control branching (for example, branching back to the start of a loop if the decremented counter value is nonzero).

- Computed GOTOs, case statements, etc.:

  Use the CTR to hold the address to branch to, and use the **bcctr** instruction with the LR option disabled (LK = 0) to branch to the selected address.

- Direct subroutine linkage—where A calls B and B returns to A. The two branches should be as follows:
  - A calls B: use a branch instruction that enables the LR (LK = 1).
  - B returns to A: use the **bclr** instruction with the LR option disabled (LK = 0) (the return address is in, or can be restored to, the LR).

- Indirect subroutine linkage:

  Where A calls Glue, Glue calls B, and B returns to A rather than to Glue. (Such a calling sequence is common in linkage code used when the subroutine that the programmer wants to call, here B, is in a different module from the caller: the binder inserts glue code to mediate the branch.) The three branches should be as follows:
  - A calls Glue: use a branch instruction that sets the LR with the link register option enabled (LK = 1).
  - Glue calls B: place the address of B in the CTR, and use the **bcctr** instruction with the link register option disabled (LK = 0).
  - B returns to A: use the **bclr** instruction with the link register option disabled (LK = 0) (the return address is in, or can be restored to, the LR.

4.2.4.3 Branch Instructions

Table 4-21 describes the branch instructions provided by the processors.

Table 4-21. Branch Instructions

Name	Mnemonic	Syntax	Operation
Branch	**b** **ba** **bl** **bla**	target_addr	**b** Branch. Branch to the address computed as the sum of the immediate address and the address of the current instruction. **ba** Branch Absolute. Branch to the absolute address specified. **bl** Branch then Link. Branch to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the link register (LR). **bla** Branch Absolute then Link. Branch to the absolute address specified. The instruction address following this instruction is placed into the LR.
Branch Conditional	**bc** **bca** **bcl** **bcla**	BO,BI,target_addr	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in Table 4-20. **bc** Branch Conditional. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. **bca** Branch Conditional Absolute. Branch conditionally to the absolute address specified. **bcl** Branch Conditional then Link. Branch conditionally to the address computed as the sum of the immediate address and the address of the current instruction. The instruction address following this instruction is placed into the LR. **bcla** Branch Conditional Absolute then Link. Branch conditionally to the absolute address specified. The instruction address following this instruction is placed into the LR.
Branch Conditional to Link Register	**bclr** **bclrl**	BO,BI	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in Table 4-20. **bclr** Branch Conditional to Link Register. Branch conditionally to the address in the LR. **bclrl** Branch Conditional to Link Register then Link. Branch conditionally to the address specified in the LR. The instruction address following this instruction is then placed into the LR.
Branch Conditional to Count Register	**bcctr** **bcctrl**	BO,BI	The BI operand specifies the bit in the CR to be used as the condition of the branch. The BO operand is used as described in Table 4-20. **bcctr** Branch Conditional to Count Register. Branch conditionally to the address specified in the count register. **bcctrl** Branch Conditional to Count Register then Link. Branch conditionally to the address specified in the count register. The instruction address following this instruction is placed into the LR. **Note:** If the “decrement and test CTR” option is specified (BO[2] = 0), the instruction form is invalid.

4.2.4.4 Simplified Mnemonics for Branch Processor Instructions

To simplify assembly language programming, a set of simplified mnemonics and symbols is provided for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions. Appendix E, “Simplified Mnemonics for PowerPC Instructions,” lists examples.

4.2.4.5 Condition Register Logical Instructions

Condition register logical instructions, shown in Table 4-22, and the Move Condition Register Field (**mcrf**) instruction are also defined as flow control instructions.

Note that if the LR update option is enabled for any of these instructions, the architecture defines these forms of the instructions as invalid.

Table 4-22. Condition Register Logical Instructions

Name	Mnemonic	Syntax	Operation
Condition Register AND	**crand**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register OR	**cror**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register XOR	**crxor**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.
Condition Register NAND	**crnand**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register NOR	**crnor**	**crb**D,**crb**A,**crb**B	The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register Equivalent	**creqv**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B. The complemented result is placed into the CR bit specified by **crb**D.
Condition Register AND with Complement	**crandc**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is ANDed with the complement of the CR bit specified by **crb**B and the result is placed into the CR bit specified by **crb**D.
Condition Register OR with Complement	**crorc**	**crb**D,**crb**A, **crb**B	The CR bit specified by **crb**A is ORed with the complement of the CR bit specified by **crb**B and the result is placed into the CR bit specified by **crb**D.
Move Condition Register Field	**mcrf**	**crf**D,**crf**S	The contents of **crf**S are copied into **crf**D. No other condition register fields are changed.

4.2.4.6 Trap Instructions

The trap instructions shown in Table 4-23 are provided to test for a specified set of conditions. If any of the conditions tested by a trap instruction are met, the system trap handler is invoked. If the tested conditions are not met, instruction execution continues normally. Appendix E, “Simplified Mnemonics for PowerPC Instructions,” lists all simplified mnemonics.

Table 4-23. Trap Instructions

Name	Mnemonic	Syntax	Description
Trap Word Immediate	**twi**	TO,**r**A,SIMM	The contents of **r**A are compared with the sign-extended SIMM operand. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.
Trap Word	**tw**	TO,**r**A,**r**B	The contents of **r**A are compared with the contents of **r**B. If any bit in the TO operand is set and its corresponding condition is met by the result of the comparison, the system trap handler is invoked.

4.2.4.7 System Linkage Instruction—UISA

Table 4-24 describes the System Call (**sc**) instruction, which permits a program to call on the system to perform a service. See Section 4.4.1, "System Linkage Instructions—OEA," for a complete description of the **sc** instruction.

Table 4-24. System Linkage Instruction—UISA

Name	Mnemonic	Syntax	Operation
System Call	**sc**	—	**sc** calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers will depend on the register conventions used by the program providing the system service. This instruction is context synchronizing as described in Section 4.1.4.1, "Context Synchronizing Instructions."

4.2.5 Processor Control Instructions—UISA

U

Processor control instructions are used to read from and write to the condition register (CR), machine state register (MSR), and special-purpose registers (SPRs). See Section 4.3.1, "Processor Control Instructions—VEA," for the **mftb** instruction and Section 4.4.2, "Processor Control Instructions—OEA," for information about the instructions used for reading from and writing to the MSR and SPRs.

V

O

4.2.5.1 Move to/from Condition Register Instructions

U

Table 4-25 summarizes the instructions for accessing the CR.

Table 4-25. Move to/from Condition Register Instructions

Name	Mnemonic	Syntax	Operation
Move to Condition Register Fields	**mtcrf**	CRM,**r**S	The contents of **r**S are placed into the CR under control of the field mask specified by operand CRM. The field mask identifies the 4-bit fields affected. Let *i* be an integer in the range 0–7. If CRM(*i*) = 1, CR field *i* (CR bits 4 * *i* through 4 * *i* + 3) is set to the contents of the corresponding field of **r**S.
Move to Condition Register from XER	**mcrxr**	**crf**D	The contents of XER[0–3] are copied into the CR field designated by **crf**D. All other CR fields remain unchanged. The contents of XER[0–3] are cleared.
Move from Condition Register	**mfcr**	**r**D	The contents of the CR are placed into **r**D.

4.2.5.2 Move to/from Special-Purpose Register Instructions (UISA)

Table 4-26 provides a brief description of the **mtspr** and **mfspr** instructions. For more detailed information refer to Chapter 8, "Instruction Set."

Table 4-26. Move to/from Special-Purpose Register Instructions (UISA)

Name	Mnemonic	Syntax	Operation
Move to Special-Purpose Register	**mtspr**	SPR,**r**S	The value specified by **r**S are placed in the specified SPR.
Move from Special-Purpose Register	**mfspr**	**r**D,SPR	The contents of the specified SPR are placed in **r**D.

4.2.6 Memory Synchronization Instructions—UISA

U Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events, and the order in which memory operations are seen by other processors or memory access mechanisms.

The number of cycles required to complete a **sync** instruction depends on system parameters and on the processor's state when the instruction is issued. As a result, frequent use of this instruction may degrade performance slightly. The **eieio** instruction may be more appropriate than **sync** for many cases.

The architecture defines the **sync** instruction with CR update enabled (Rc field, bit 31 = 1) to be an invalid form.

The proper paired use of the **lwarx** with **stwcx.** instructions allows programmers to emulate common
V semaphore operations such as test and set, compare and swap, exchange memory, and fetch and add.
U Examples of these semaphore operations can be found in Appendix D, “Synchronization Programming Examples.” The **lwarx** instruction must be paired with an **stwcx.** instruction with the same effective address specified by both instructions of the pair. The only exception is that an unpaired **stwcx.** instruction to any (scratch) effective address can be used to clear any reservation held by the processor. Note reservation granularity is implementation-dependent.

The **lwarx** and **stwcx.** instructions allow a processor to load a semaphore from memory, compute a result based on the value of the semaphore, and conditionally store it back to the same location. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx**. If the reservation exists when the store is executed, the store is performed and a CR bit is set. If the reservation does not exist when the store is executed, the target memory location is not modified and a CR bit is cleared.

The **lwarx** and **stwcx.** primitives allow software to read a semaphore, compute a result based on the value of the semaphore, store the new value back into the semaphore location only if that location has not been modified since it was first read, and determine if the store was successful. If the store was successful, the sequence of instructions from the read of the semaphore to the store that updated the semaphore appear to have been executed atomically (that is, no other processor or mechanism modified the semaphore location between the read and the update), thus providing the equivalent of a real atomic operation. However, in reality, other processors may have read from the location during this operation.

The **lwarx** and **stwcx.** instructions require the EA to be aligned.

In general, the **lwarx** and **stwcx.** instructions should be used only in system programs, which can be invoked by application programs as needed.

At most one reservation exists simultaneously on any processor. The address associated with the reservation can be changed by a subsequent **lwarx**. The conditional store is performed based upon the existence of a reservation established by the preceding **lwarx**.

A reservation held by the processor is cleared (or may be cleared, in the case of the fourth and fifth bullet items) by one of the following:

- The processor holding the reservation executes another **lwarx**; this clears the first reservation and establishes a new one.

- The processor holding the reservation executes any **stwcx.** regardless of whether its address matches that of the **lwarx**.
- Some other processor executes a store or **dcbz** to the same reservation granule or modifies a reference or change bit in the same reservation granule.
- Some other processor executes a **dcbtst**, **dcbst**, **dcbf**, or **dcbi** to the same reservation granule; regarless of whether the reservation is cleared is undefined.
- Some other processor executes a **dcba** to the same reservation granule. The reservation is cleared if the instruction causes the target block to be newly established in the data cache or to be modified; otherwise, whether the reservation is cleared is undefined.
- Some other mechanism modifies a memory location in the same reservation granule.

  Note that interrupts do not clear reservations; however, system software invoked by interrupts may clear reservations.

Table 4-27 summarizes the memory synchronization instructions defined by the UISA. Section 4.3.2, "Memory Synchronization Instructions—VEA," summarizes additional memory synchronization instructions (**eieio** and **isync**). U

Table 4-27. Memory Synchronization Instructions—UISA

Name	Mnemonic	Syntax	Operation
Load Word and Reserve Indexed	**lwarx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by the EA is loaded into **r**D.
Store Word Conditional Indexed	**stwcx.**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If a reservation exists and the EA specified by **stwcx.** is the same as that specified by the load and reserve instruction that established the reservation, the contents of **r**S are stored into the word addressed by the EA and the reservation is cleared. If a reservation exists but the EA specified by **stwcx.** is not the same as that specified by the **lwarx** that established the reservation, the reservation is cleared, and it is undefined whether the contents of **r**S are stored into the word in memory addressed by the EA. If no reservation exists, the instruction completes without altering memory or the cache contents.
Synchronize	**sync**	—	Executing a **sync** instruction ensures that all instructions preceding the **sync** instruction appear to have completed before the **sync** instruction completes, and that no subsequent instructions are initiated by the processor until after the **sync** instruction completes. When the **sync** instruction completes, all memory accesses caused by instructions preceding the **sync** instruction will have been performed with respect to all other mechanisms that access memory. See Chapter 8, "Instruction Set," for more information.

4.3 VEA Instructions

The virtual environment architecture (VEA) describes the semantics of the memory model that can be assumed by software processes, and includes descriptions of the cache model, cache-control instructions, address aliasing, and other related functionality. Implementations that conform to the VEA also adhere to the UISA, but may not necessarily adhere to the OEA. V O U

This section describes additional instructions that are provided by the VEA.

4.3.1 Processor Control Instructions—VEA

The VEA defines the **mftb** instruction (user-level instruction) for reading the contents of the time base register; see Section 2.2, "VEA Register Set—Time Base," for more information. Table 4-28 describes the **mftb** instruction.

Table 4-28. Move from Time Base Instruction

Name	Mnemonic	Syntax	Operation
Move from Time Base	**mftb**	**r**D, TBR	The TBR field denotes either time base lower or time base upper, encoded as shown in Table 4-29 and Table 4-30. The contents of the designated register are copied to **r**D. Simplified mnemonics are provided (See Section E.8, "Simplified Mnemonics for Accessing SPRs") so **mftb** can be coded with the TBR name as part of the mnemonic rather than as an operand. The simplified mnemonics Move from Time Base (**mftb**) and Move from Time Base Upper (**mftbu**) are variants of **mftb** rather than of the **mfspr** instruction. The **mftb** instruction serves as both a basic and simplified mnemonic. Assemblers recognize an **mftb** mnemonic with two operands as the basic form, and an **mftb** mnemonic with one operand as the simplified form. On 32-bit implementations, it is not possible to read the entire 64-bit time base register in a single instruction. The **mftb** simplified mnemonic moves from the lower half of the time base register (TBL) to a GPR, and the **mftbu** simplified mnemonic moves from the upper half of the time base (TBU) to a GPR.

Table 4-29 summarizes the time base (TBL/TBU) register encodings to which user-level access (using **mftb**) is permitted (as specified by the VEA).

Table 4-29. User-Level TBR Encodings (VEA)

Decimal Value in TBR Field	TBR[0–4] TBR[5–9]	Register Name	Description
268	01100 01000	TBL	Time base lower (read-only)
269	01101 01000	TBU	Time base upper (read-only)

Table 4-30 summarizes the TBL and TBU register encodings to which supervisor-level access (using **mtspr**) is permitted.

Table 4-30. Supervisor-Level TBR Encodings (VEA)

Decimal Value in SPR Field	SPR[0–4] SPR[5–9]	Register Name	Description
284	11100 01000	TBL[1]	Time base lower (write only)
285	11101 01000	TBU[1]	Time base upper (write only)

[1] Moving from the time base (TBL and TBU) can also be accomplished with the **mftb** instruction.

4.3.2 Memory Synchronization Instructions—VEA

Memory synchronization instructions control the order in which memory operations are completed with respect to asynchronous events and the order in which memory operations are seen by other processors or memory access mechanisms. Section 4.1.4, “Synchronizing Instructions,” describes these instructions and related aspects of memory synchronization.

System designs that use a second-level cache should take special care to recognize the hardware signaling caused by a **sync** operation and perform the appropriate actions to guarantee that memory references that may be queued internally to the second-level cache have been performed globally.

In addition to the **sync** instruction (specified by UISA), the VEA defines the Enforce In-Order Execution of I/O (**eieio**) and Instruction Synchronize (**isync**) instructions; see Table 4-31. The number of cycles required to complete an **eieio** instruction depends on system parameters and on the processor's state when the instruction is issued. As a result, frequent use of this instruction may degrade performance slightly.

The **isync** instruction causes the processor to wait for any preceding instructions to complete, discard all prefetched instructions, and then branch to the next sequential instruction (which has the effect of clearing the pipeline behind the **isync** instruction).

Table 4-31. Memory Synchronization Instructions—VEA

Name	Mnemonic	Syntax	Operation
Enforce In-Order Execution of I/O	**eieio**	—	Provides an ordering function for the effects of loads and stores executed by a processor. See also Section 5.2.1.1, “Enforce In-Order Execution of I/O Instruction (eieio).”
Instruction Synchronize	**isync**	—	Executing an **isync** ensures that all previous instructions complete before the **isync** completes, although memory accesses caused by those instructions need not have been performed with respect to other processors and mechanisms. It also ensures that the processor initiates no subsequent instructions until the **isync** completes. Finally, it causes the processor to discard any prefetched instructions, so subsequent instructions are fetched and executed in the context established by the instructions preceding the **isync**. This instruction does not affect other processors or their caches.

4.3.3 Memory Control Instructions—VEA

Memory control instructions include Cache management instructions (user-level and supervisor-level), and Translation lookaside buffer (TLB) management instructions.

This section describes the user-level cache management instructions defined by the VEA. Section 4.4.3, “Memory Control Instructions—OEA,” describes the remaining memory control instructions including supervisor-level cache, segment register manipulation, and translation lookaside buffer management instructions.

4.3.3.1 User-Level Cache Instructions—VEA

The instructions summarized in this section provide user-level programs the ability to manage on-chip caches if they are implemented. See Section 5.2.5, "VEA Cache Management Instructions," for more information about cache instructions.

As with other memory-related instructions, the effect of the cache management instructions on memory are weakly ordered. If the programmer needs to ensure that cache or other instructions have been performed with respect to all other processors and system mechanisms, a **sync** instruction must be placed in the program following those instructions.

Note that when data address translation is disabled (MSR[DR] = 0), the Data Cache Block Clear to Zero (**dcbz**) and the Data Cache Block Allocate (**dcba**) instructions allocate a cache block in the cache and may not verify that the physical address (referred to as real address in the architecture specification) is valid. If a cache block is created for an invalid physical address, a machine check condition may result when an attempt is made to write that cache block back to memory. The cache block could be written back as a result of the execution of an instruction that causes a cache miss and the invalid addressed cache block is the target for replacement or a Data Cache Block Store (**dcbst**) instruction.

Any cache control instruction that generates an effective address for which SR[T] = 1 is treated as a no-op.

Table 4-32 summarizes the cache instructions defined by the VEA. Note that these instructions are accessible to user-level programs.

Table 4-32. User-Level Cache Instructions

Name	Mnemonic	Syntax	Operation
Data Cache Block Touch	**dcbt**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). **dcbt** instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon load from the addressed byte. See Section 5.2.5.1.1, "Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst) Instructions."
Data Cache Block Touch for Store	**dcbtst**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). **dcbtst** instruction is a hint that performance will probably be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store into the addressed byte.

Table 4-32. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Operation
Data Cache Block Allocate	**dcba**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If the cache block containing the addressed byte is in the data cache, all bytes of the cache block are made undefined, but the cache block remains valid. Note that programming errors can occur if the data in this cache block is subsequently read or used inadvertently. If the page containing the addressed byte is not in the data cache and the corresponding page is marked caching allowed (I = 0), the cache block is allocated (and made valid) in the data cache without fetching the block from main memory, and the value of all bytes of the cache block is undefined. If the page containing the addressed byte is marked caching inhibited (WIM = x1x), this instruction is treated as a no-op. If the addressed cache block is located in a page marked as memory coherent (WIM = xx1) and the cache block exists in the caches of other processors, memory coherence is maintained in those caches. **dcba** is treated as a store to the addressed byte with respect to address translation, memory protection, reference and change recording, and the ordering enforced by **eieio** or by the combination of caching-inhibited and guarded attributes for a page. This instruction is optional. (In the OEA, **dcba** is additionally defined to clear all bytes of a newly established block if that block did not exist in the cache.)
Data Cache Block Clear to Zero	**dcbz**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If the cache block containing the addressed byte is in the data cache, all bytes of the cache block are cleared to zero. If the page containing the byte addressed by the EA is not in the data cache and the corresponding page is marked caching allowed (I = 0), the cache block is established in the data cache without fetching the block from main memory, and all bytes of the cache block are cleared to zero. If the page containing the byte addressed by the EA is marked caching inhibited (WIM = x1x) or write-through (WIM = 1xx), either all bytes of the area of main memory that corresponds to the addressed cache block are cleared to zero, or an alignment interrupt occurs. If the cache block addressed by the EA is located in a page marked as memory coherent (WIM = xx1) and the cache block exists in the caches of other processors, memory coherence is maintained in those caches. The **dcbz** instruction is treated as a store to the addressed byte with respect to address translation, memory protection, reference and change recording, and the ordering enforced by **eieio** or by the combination of caching-inhibited and guarded attributes for a page. See Section 5.2.5.1.2, “Data Cache Block Set to Zero (dcbz) Instruction.”
Data Cache Block Store	**dcbst**	**r**A,**r**B	The EA is the sum(**r**A|0) + (**r**B). If the cache block containing the addressed byte is located in a page marked memory coherent (WIM = xx1), and a cache block containing the addressed byte is in the data cache of any processor and has been modified, the cache block is written to main memory. If the cache block containing the byte addressed by the EA is located in a page not marked memory coherent (WIM = xx0), and a cache block containing the byte addressed by EA is in the data cache of this processor and has been modified, the cache block is written to main memory. The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed settings of the cache block containing the byte addressed by the EA. The **dcbst** instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for reference and change bit recording except that reference and change bit recording may not occur**.** Note that some implementations may execute this instruction as a **dcbf**. This instruction is optional. See Section 5.2.5.1.1, “Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst) Instructions.”

Table 4-32. User-Level Cache Instructions (continued)

Name	Mnemonic	Syntax	Operation
Data Cache Block Flush	**dcbf**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The action taken depends on the memory mode associated with the target, and on the state of the block. The following list describes the action taken for the various cases, regardless of whether the page or block containing the addressed byte is designated as write-through or if it is caching-inhibited or caching-allowed. • Coherency required (WIM = xx1) — Unmodified block—Invalidates copies of the block in the caches of all processors. — Modified block—Copies the block to memory. Invalidates copies of the block in the caches of all processors. — Absent block—If modified copies of the block are in the caches of other processors, causes them to be copied to memory and invalidated. If unmodified copies are in the caches of other processors, causes those copies to be invalidated. • Coherency not required (WIM = xx0) — Unmodified block—Invalidates the block in the processor's cache. — Modified block—Copies the block to memory. Invalidates the block in the processor's cache. — Absent block—Does nothing. The function **dcbf** is independent of the write-through/write-back and caching-inhibited/caching-allowed settings for the cache block containing the addressed byte. The **dcbf** is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for reference and change bit recording except that reference and change bit recording may not occur. See Section 5.2.5.1.4, "Data Cache Block Flush (dcbf) Instruction."
Instruction Cache Block Invalidate	**icbi**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). If the cache block containing the addressed byte in a page marked memory coherent (WIM = xx1), and a cache block containing addressed byte is in the instruction cache of any processor, the cache block is made invalid in all such caches, so that the next reference causes the cache block to be refetched. If the cache block containing the addressed byte is located in a page not marked memory coherent (WIM = xx0), and a cache block containing the addressed byte is in the instruction cache of this processor, the cache block is made invalid in that cache, so that the next reference causes the block to be refetched. The function of **icbi** is independent of the write-through/write-back and caching-inhibited/caching-allowed modes of the cache block containing the byte addressed by the EA. **icbi** is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for reference and change bit recording except that reference and change bit recording may not occur. See Section 5.2.5.2.1, "Instruction Cache Block Invalidate Instruction (icbi)."

4.3.4 External Control Instructions (Optional)

The optional external control instructions allow a user-level program to communicate with a special-purpose device. Two instructions are provided and are summarized in Table 4-33.

Table 4-33. External Control Instructions

Name	Mnemonic	Syntax	Operation
External Control In Word Indexed	**eciwx**	**r**D,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B) and must be word-aligned. A load word request for the physical address corresponding to the EA is sent to the device identified by the EAR[RID] (bits 26–31), bypassing the cache. The word returned by the device is placed into **r**D. **eciwx** is treated as a load from the addressed byte with respect to address translation, memory protection, reference and change recording, and the ordering performed by **eieio**.
External Control Out Word Indexed	**ecowx**	**r**S,**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B) and must be word-aligned. A store word request for the physical address corresponding to the EA and the contents of **r**S are sent to the device identified by EAR[RID], bypassing the cache. The EA sent to the device must be word-aligned. **ecowx** is treated as a store to the addressed byte with respect to address translation, memory protection, reference and change recording, and the ordering performed by **eieio**. Software synchronization is required in order to ensure that the data access is performed in program order with respect to data accesses caused by other store or **ecowx** instructions, even though the addressed byte is assumed to be caching-inhibited and guarded.

4.4 OEA Instructions

O

The operating environment architecture (OEA) includes the structure of the memory management model, supervisor-level registers, and the interrupt model. Implementations that conform to the OEA also adhere to the UISA and the VEA. This section describes the instructions provided by the OEA. All instructions listed are supervisor level.

V

U

4.4.1 System Linkage Instructions—OEA

O

This section describes the system linkage instructions (see Table 4-34). The **sc** instruction is a user-level instruction that permits a user program to call on the system to perform a service and causes the processor to take an interrupt.

Table 4-34. System Linkage Instructions—OEA

Name	Mnemonic	Syntax	Operation
System Call	**sc**	—	When executed, the effective address of the instruction following the **sc** instruction is placed into SRR0. SRR1[1–4,10–15] are cleared. Additionally, 16–23, 25–27, and 30–31are placed into the corresponding SRR1bits. Depending on the implementation, additional MSR bits may also be saved in SRR1. Then a system call interrupt is generated. The interrupt causes the MSR to be altered as described in Section 6.5, "Interrupt Definitions." The interrupt causes the next instruction to be fetched from offset 0xC00 from the base physical address indicated by the new setting of MSR[IP]. This instruction is context synchronizing.
Return from Interrupt	**rfi**	—	[6–23, 25–27, 30–31] are placed into the corresponding MSR bits. Depending on the implementation, additional bits of MSR may also be restored from SRR1. If the new MSR value does not enable any pending interrupts, the next instruction is fetched, under control of the new MSR value, from the address SRR0[0–29] || 0b00. If the new MSR value enables one or more pending interrupts, the interrupt associated with the highest priority pending interrupt is generated; in this case the value placed into SRR0 (machine status save/restore 0) by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred. This context-synchronizing instruction is defined only for 32-bit implementations.

4.4.2 Processor Control Instructions—OEA

This section describes the processor control instructions that are used to read from and write to the MSR and the SPRs.

4.4.2.1 Move to/from Machine State Register Instructions

Table 4-35 summarizes the instructions used for reading from and writing to the MSR.

Table 4-35. Move to/from Machine State Register Instructions

Name	Mnemonic	Syntax	Operation
Move to Machine State Register	**mtmsr**	**r**S	The contents of **r**S are placed into the MSR. This supervisor-level instruction is context synchronizing except with respect to alterations to the POW and LE bits. See Section 2.3.17, "Synchronization Requirements for Special Registers and for Lookaside Buffers."
Move from Machine State Register	**mfmsr**	**r**D	The MSR contents of the are placed into **r**D.

4.4.2.2 Move to/from Special-Purpose Register Instructions (OEA)

Provided is a brief description of the **mtspr** and **mfspr** instructions (see Table 4-36). For more detailed information, see Chapter 8, "Instruction Set." Simplified mnemonics are provided for these instructions in Appendix E, "Simplified Mnemonics for PowerPC Instructions." For a discussion of context synchronization requirements when altering certain SPRs, refer to Appendix D, "Synchronization Programming Examples."

Table 4-36. Move to/from Special-Purpose Register Instructions (OEA)

Name	Mnemonic	Syntax	Operation
Move to Special-Purpose Register	**mtspr**	SPR,**r**S	The contents of **r**S are placed into the designated SPR. For 32-bit SPRs, the contents of **r**S are placed into the SPR. For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.
Move from Special-Purpose Register	**mfspr**	**r**D,SPR	The contents of the designated SPR are placed into **r**D.

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction encoding, with the high-order 5 bits appearing in bits 16–20 of the instruction encoding and the low-order 5 bits in bits 11–15.

For information on SPR encodings (both user- and supervisor-level), see Chapter 8, “Instruction Set.” Note that there are additional SPRs specific to each implementation; for implementation-specific SPRs, see the user’s manual for that particular processor.

4.4.3 Memory Control Instructions—OEA

Memory control instructions include the following types of instructions:

- Cache management instructions (supervisor-level and user-level)
- Segment register manipulation instructions
- Translation lookaside buffer (TLB) management instructions

This section describes supervisor-level memory control instructions. See Section 4.3.3, “Memory Control Instructions—VEA,” for more information about user-level cache management instructions.

4.4.3.1 Supervisor-Level Cache Management Instruction

Table 4-37 summarizes the operation of the only supervisor-level cache management instruction, data cache block invalidate (**dcbi**). See Section 4.3.3.1, “User-Level Cache Instructions—VEA,” for user-level cache instructions.

Note that any cache control instruction that generates an effective address for which SR[T] = 1 is treated as a no-op.

Table 4-37. Cache Management Supervisor-Level Instruction

Name	Mnemonic	Syntax	Operation
Data Cache Block Invalidate	**dcbi**	**r**A,**r**B	The EA is the sum (**r**A|0) + (**r**B). The action taken depends on the memory mode associated with the target, and the state (modified, unmodified) of the cache block. The following list describes the action to take if the cache block containing the byte addressed by the EA is or is not in the cache. • Coherency required (WIM = xx1) — Unmodified cache block—Invalidates copies of the cache block in the caches of all processors. — Modified cache block—Invalidates copies of the cache block in the caches of all processors. (Discards the modified contents.) — Absent cache block—If copies are in the caches of any other processor, causes the copies to be invalidated. (Discards any modified contents.) • Coherency not required (WIM = xx0) — Unmodified cache block—Invalidates the cache block in the local cache. — Modified cache block—Invalidates the cache block in the local cache. (Discards the modified contents.) — Absent cache block—No action is taken. When data address translation is enabled, MSR[DT]=1, and the logical (effective) address has no translation, a data access interrupt occurs. The function of this instruction is independent of the write-through and cache-inhibited/allowed modes determined by the WIM bit settings of the block containing the byte addressed by the EA. This instruction is treated as a store to the addressed byte with respect to address translation and protection, except that the change bit need not be set, and if the change bit is not set then the reference bit need not be set. Note that some implementations may execute this instruction as a **dcbf**. This instruction is optional.

4.4.3.2 Segment Register Manipulation Instructions

The instructions listed in Table 4-38 provide access to the segment registers (SRs). These instructions operate independently of the MSR[IR] and MSR[DR] bit settings. Refer to Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for serialization requirements and other recommended precautions to observe when manipulating the SRs.

Table 4-38. Segment Register Manipulation Instructions

Name	Mnemonic	Syntax	Operation
Move to Segment Register	**mtsr**	SR,**r**S	The contents of **r**S are placed into the specified SR.
Move to Segment Register Indirect	**mtsrin**	**r**S,**r**B	The contents of **r**S are copied to the SR selected by bits 0–3 of **r**B.
Move from Segment Register	**mfsr**	**r**D,SR	The contents specified SR are placed into **r**D.
Move from Segment Register Indirect	**mfsrin**	**r**D,**r**B	The contents selected by bits 0–3 of **r**B are copied into **r**D.

4.4.3.3 Translation Lookaside Buffer Management Instructions

The address translation mechanism is defined in terms of segment descriptors and page table entries (PTEs) used to locate the logical-to-physical address mapping for a particular access. These segment descriptors and PTEs reside in segment tables and page tables in memory, respectively.

For performance reasons, many processors implement one or more translation lookaside buffers on-chip. These are caches of portions of the page table. As changes are made to the address translation tables, it is necessary to maintain coherency between the TLB and the updated tables. This is done by invalidating TLB entries or by invalidating the entire TLB and allowing the translation caching mechanism to refetch from the tables.

Each implementation that has a TLB provides means for invalidating an individual TLB entry and invalidating the entire TLB.

If a processor does not implement a TLB, it treats the corresponding instructions (**tlbie**, **tlbia**, and **tlbsync**) either as no-ops or as illegal instructions.

Refer to Chapter 7, “Memory Management,” for more information about TLB operation. Table 4-39 summarizes the operation of the TLB instructions.

Table 4-39. Translation Lookaside Buffer Management Instructions

Name	Mnemonic	Syntax	Operation
TLB Invalidate Entry	**tlbie**	**r**B	The EA is the contents of **r**B. If the TLB contains an entry corresponding to the EA, that entry is removed from the TLB. The TLB search is performed regardless of the MSR[IR,DR] settings. Any block address translation for the EA is ignored. This instruction causes the target TLB entry to be invalidated in all processors. The operation performed by this instruction is treated as a caching inhibited and guarded data access with respect to the ordering performed by **eieio**. Instruction is optional.
TLB Invalidate All	**tlbia**	—	All TLB entries are invalidated, regardless of the MSR[IR,DR] settings. This instruction does not cause the entries to be invalidated in other processors. This instruction is optional.
TLB Synchronize	**tlbsync**	—	Ensures that all **tlbie** instructions previously executed by the processor executing **tlbsync** have completed on all processors. The operation performed by this instruction is treated as a caching-inhibited and guarded data access with respect to the ordering performed by **eieio**. This instruction is optional.

Because the presence and exact semantics of TLB management instructions is implementation-dependent, system software should incorporate uses of the instruction into subroutines to minimize compatibility problems.

Chapter 5
Cache Model and Memory Coherency

This chapter summarizes the cache model as defined by the virtual environment architecture (VEA) as well as the built-in architectural controls for maintaining memory coherency. It also further describes the cache control instructions and special concerns for memory coherency in single-processor and multiprocessor systems. Aspects of the operating environment architecture (OEA) as they relate to the cache model and memory coherency are also covered.

5.1 Overview

The PowerPC architecture provides for relaxed memory coherency. Features such as write-back caching and memory access reordering allow software engineers to exploit the performance benefits of weakly-ordered memory access. The architecture also provides the means to control the order of accesses for order-critical operations.

In this chapter, the term multiprocessor is used in the context of maintaining cache coherency. In this context, a system could include other devices that access system memory, maintain independent caches, and function as bus masters.

Each cache management instruction operates on an aligned unit of memory. The VEA defines this cacheable unit as a block. This chapter uses the term ‘cache block’ to distinguish it from the unit of memory addressed by the block address translation (BAT) mechanism. The cache block size can vary by implementation. In addition, the unit of memory at which coherency is maintained is called the coherence block, the size of which is also implementation-specific, although it is typically the same size as a cache block.

5.2 The Virtual Environment

The user instruction set architecture (UISA) relies upon a memory space of 2^{32} bytes for applications. The VEA expands upon the memory model by introducing virtual memory, caches, and shared memory multiprocessing. Although many applications do not need to access the features introduced by the VEA, it is important that programmers be aware that they are working in a virtual environment where the physical memory may be shared by multiple processes running on one or more processors.

This section describes load and store ordering, atomicity, the cache model, memory coherency, and the VEA cache management instructions. VEA features are accessible to both user- and supervisor-level applications (referred to as problem state and privileged state, respectively, in the architecture specification).

The mechanism for controlling the virtual memory space is defined by the OEA. OEA features are accessible to supervisor-level applications only (typically operating systems). For more information on address translation, refer to Chapter 7, “Memory Management.”

5.2.1 Memory Access Ordering

The VEA specifies a weakly consistent memory model for shared memory multiprocessor systems. This model provides an opportunity for significantly improved performance over a model that has stronger consistency rules, but places the responsibility for access ordering on the programmer. When a program requires strict access ordering for proper execution, the programmer must insert the appropriate ordering or synchronization instructions into the program.

The order in which the processor performs memory accesses, the order in which those accesses complete in memory, and the order in which those accesses are viewed as occurring by another processor may all be different. A means of enforcing memory access ordering is provided to allow programs (or instances of programs) to share memory. Similar means are needed to allow programs executing on a processor to share memory with some other mechanism, such as an I/O device, that can also access memory.

Various facilities are provided that enable programs to control the order in which memory accesses are performed by separate instructions. First, if separate store instructions access memory that is designated as both caching-inhibited and guarded, the accesses are performed in the order specified by the program. Refer to Section 5.2.4, “Memory Coherency,” and Section 5.3.1, “Memory/Cache Access Attributes,” for a complete description of the caching-inhibited and guarded attributes. Additionally, two instructions, **eieio** and **sync**, are provided that enable the program to order memory accesses caused by separate instructions.

No ordering should be assumed among the memory accesses caused by a single instruction (that is, by an instruction for which multiple accesses are not atomic), and no means are provided for controlling that order. Chapter 8, “Instruction Set,” contains additional information about **sync** and **eieio**.

5.2.1.1 Enforce In-Order Execution of I/O Instruction (eieio)

The **eieio** instruction described in Section 4.3.2, “Memory Synchronization Instructions—VEA,” creates a memory barrier, which provides an ordering function for the memory accesses caused by load, store, **dcbz**, and **dcba** instructions executed by the processor executing the **eieio** instruction. These accesses are divided into two sets, which are ordered separately. The access caused by **dcbz** or **dcba** is ordered as a store.

The **eieio** instruction permits the program to control the order in which loads and stores are performed when the accessed memory has certain attributes, as described in Chapter 8, “Instruction Set.” For example, **eieio** can be used to ensure that a sequence of load and store operations to an I/O device’s control registers updates those registers in the desired order. The **eieio** instruction can also be used to ensure that all stores to a shared data structure are visible to other processors before the store that releases the lock is visible to them.

The **eieio** instruction may complete before memory accesses caused by instructions preceding the **eieio** have been performed.

If stronger ordering is desired, **sync** must be used.

5.2.1.2 Synchronize Instruction

When a portion of memory that requires coherency must be forced to a known state, it is necessary to synchronize memory with respect to other processors and mechanisms. This synchronization is accomplished by requiring programs to indicate explicitly in the instruction stream, by inserting a **sync**, that synchronization is required. Only when **sync** completes are the effects of all coherent memory accesses previously executed by the program guaranteed to have been performed with respect to all other processors and mechanisms that access those locations coherently. The **sync** instruction is described in Chapter 8, “Instruction Set.”

5.2.2 Atomicity

An access is atomic if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized—each happens in its entirety in some order, even when that order is neither specified in the program nor enforced between processors.

Only the following single-register accesses are guaranteed to be atomic:

- Byte accesses (all bytes are aligned on byte boundaries)
- Half-word accesses aligned on half-word boundaries
- Word accesses aligned on word boundaries

No other accesses are guaranteed to be atomic. In particular, the accesses caused by the following instructions are not guaranteed to be atomic:

- Load and store instructions with misaligned operands
- Load multiple and string instructions: **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx**
- Floating-point double-word accesses in 32-bit implementations
- Any cache management instructions

The **lwarx**/**stwcx.** instruction combinations can be used to perform atomic memory references. The **lwarx** is a load from a word-aligned location that has two side effects:

1. A reservation for a subsequent **stwcx.** is created.
2. The memory coherence mechanism is notified that a reservation exists for the memory location accessed by the **lwarx**.

The **stwcx.** is a store to a word-aligned location that is conditioned on the existence of the reservation created by **lwarx** and on whether the same memory location is specified by both instructions and whether the instructions are issued by the same processor.

In a multiprocessor system, every processor (other than the one executing **lwarx**/**stwcx.**) that might update the location must configure the addressed page as memory coherency required. The **lwarx**/**stwcx.** instructions function in caching-inhibited, as well as in caching-allowed, memory. If the addressed memory is in write-through mode, it is implementation-dependent whether these instructions function correctly or cause the DSI interrupt handler to be invoked.

The **lwarx**/**stwcx.** combination is described in Section 4.3.2, “Memory Synchronization Instructions—VEA,” and Chapter 8, “Instruction Set.”

5.2.3 Cache Model

The PowerPC architecture does not specify the type, organization, implementation, or even the existence of a cache. The standard cache model has separate instruction and data caches, also known as a Harvard cache model. However, the architecture allows for many different cache types. Some implementations may have a unified cache (where there is a single cache for both instructions and data). Other implementations may not have a cache at all.

The function of the cache management instructions depends on the cache implementation and the setting of the memory/cache access modes. For a program to execute properly on all implementations, software should use the Harvard model. In cases where a processor is implemented without a cache, the architecture guarantees that instructions affecting the nonimplemented cache does not halt execution (note that **dcbz** may cause an alignment interrupt on some implementations). For example, a processor with no cache may treat a cache instruction as a no-op. Or, a processor with a unified cache may treat **icbi** as a no-op. In this manner, programs written for separate instruction and data caches can run on all compliant implementations.

5.2.4 Memory Coherency

The primary objective of a coherent memory system is to provide the same image of memory to all devices using the system. The VEA and OEA define coherency controls that facilitate synchronization, cooperative use of shared resources, and task migration among processors. These controls include the memory/cache access attributes, the **sync** and **eieio** instructions, and the **lwarx/stwcx.** pair. Without these controls, the processor could not support a weakly-ordered memory access model.

A strongly-ordered memory access model hinders performance by requiring excessive overhead, particularly in multiprocessor environments. For example, a processor performing a store operation in a strongly-ordered system requires exclusive access to an address before making an update, to prevent another device from using stale data.

The VEA defines a page as a unit of memory for which protection and control attributes are independently specifiable. The OEA (supervisor level) specifies the size of a page as 4 Kbytes. It is important to note that the VEA (user level) does not specify the page size.

5.2.4.1 Memory/Cache Access Modes

The OEA defines the set of memory/cache access modes and the mechanism to implement these modes. Refer to Section 5.3.1, “Memory/Cache Access Attributes,” for more information. However, the VEA specifies that at the user level, the operating system can be expected to provide the following attributes for each page of memory:

- Write-through or write-back
- Caching-inhibited or caching-allowed
- Memory coherency required or memory coherency not required
- Guarded or not guarded

User-level programs specify the memory/cache access attributes through an operating system service.

5.2.4.1.1 Pages Designated as Write-Through

When a page is designated as write-through, store operations update the data in the cache and also update the data in main memory. The processor writes to the cache and through to main memory. Load operations use the data in the cache, if it is present.

In write-back mode, the processor is required only to update data in the cache. The processor may (but is not required to) update main memory. Load and store operations use the data in the cache, if it is present. The data in main memory does not necessarily stay consistent with that same location's data in the cache. Many implementations automatically update main memory in response to a memory access by another device (for example, a snoop hit). In addition, **dcbst** and **dcbf** can explicitly force an update of main memory.

The write-through attribute is meaningless for locations designated as caching-inhibited.

5.2.4.1.2 Pages Designated as Caching-Inhibited

When a page is designated as caching-inhibited, the processor bypasses the cache and performs load and store operations to main memory. When a page is designated as caching-allowed, the processor uses the cache and performs load and store operations to the cache or main memory depending on the other memory/cache access attributes for the page.

It is important that all locations in a page are purged from the cache before changing the memory/cache access attribute for the page from caching-allowed to caching-inhibited. It is considered a programming error if a caching-inhibited memory location is found in the cache. Software must ensure that the location has not previously been brought into the cache, or, if it has, that it has been flushed from the cache. If the programming error occurs, the result of the access is boundedly undefined.

5.2.4.1.3 Pages Designated as Memory Coherency Required

When a page is designated as memory coherency required, store operations to that location are serialized with all stores to that same location by all other processors that also access the location coherently.This can be implemented, for example, by an ownership protocol that allows at most one processor at a time to store to the location. Moreover, the current copy of a cache block that is in this mode may be copied to main storage any number of times, for example, by successive **dcbst** instructions.

Coherency does not ensure that the result of a store by one processor is visible immediately to all other processors and mechanisms. Only after a program has executed a **sync** are the previous accesses it executed guaranteed to have been performed with respect to all other processors and mechanisms.

5.2.4.1.4 Pages Designated as Memory Coherency Not Required

For a memory area that is configured such that coherency is not required, software must ensure that the data cache is consistent with main storage before changing the mode or allowing another device to access the area.

Executing a **dcbst** or **dcbf** specifying a cache block that is in this mode causes the block to be copied to main memory if and only if the processor modified the contents of a location in the block and the modified contents have not been written to main memory.

In a single-cache system, correct coherent execution may likely not require memory coherency; therefore, using memory coherency not required mode improves performance.

5.2.4.1.5 Pages Designated as Guarded

The guarded attribute pertains to speculative execution. Refer to Section 5.3.1.5.4, “Speculative Accesses to Guarded Memory,” for more information. Note that the term ‘speculative’ is referred to as out-of-order in the architecture specification. The use of these terms in this manual is described in Section 5.3.1.5.1, “Definition of Speculative and Out-of-Order Memory Accesses.”

Instructions and data cannot be accessed speculatively from a page designated as guarded. Additionally, if separate store instructions access memory that is caching-inhibited and guarded, accesses are performed in the order specified by the program. When a page is designated as not guarded, speculative fetches and accesses are allowed.

5.2.4.2 Coherency Precautions

Mismatched memory/cache attributes cause coherency paradoxes in both single- and multiple-processor systems. When the memory/cache access attributes are changed, it is critical that the cache contents reflect the new attribute settings. For example, if a block or page that had allowed caching becomes caching-inhibited, the appropriate cache blocks should be flushed to leave no indication that caching had previously been allowed.

Although coherency paradoxes are considered programming errors, specific implementations may attempt to handle the such conditions to minimize the negative effects on memory coherency. Bus operations generated for specific instructions and state conditions are not defined by the architecture.

5.2.5 VEA Cache Management Instructions

The VEA defines instructions for controlling both the instruction and data caches. For implementations that have a unified instruction/data cache, instruction cache control instructions are valid instructions, but may function differently.

Note that any cache control instruction that generates an EA that corresponds to a direct-store segment (SR[T] = 1) is treated as a no-op. However, the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

This section briefly describes the cache management instructions available to programs at the user privilege level. Additional descriptions of coding the VEA cache management instructions is provided in Section 4.3.3, “Memory Control Instructions—VEA,” and Chapter 8, “Instruction Set.” In the following instruction descriptions, the target is the cache block containing the byte addressed by the effective address.

5.2.5.1 Data Cache Instructions

Data caches and unified caches must be consistent with other caches (data or unified), memory, and I/O data transfers. To ensure consistency, aliased effective addresses (two effective addresses that map to the same physical address) must have the same page offset. Note that physical address is referred to as real address in the architecture specification.

5.2.5.1.1 Data Cache Block Touch (dcbt) and Data Cache Block Touch for Store (dcbtst) Instructions

These instructions provide a method for improving performance through the use of software-initiated prefetch hints. However, these instructions do not guarantee that a cache block will be fetched.

A program uses **dcbt** to request a cache block fetch before it is needed by the program. The program can then use the data from the cache rather than fetching from main memory.

The **dcbtst** instruction behaves similarly to **dcbt**. A program uses **dcbtst** to request a cache block fetch to guarantee that a subsequent store will be to a cached location.

The processor does not invoke the interrupt handler for translation or protection violations caused by either of the touch instructions. Additionally, memory accesses caused by these instructions are not necessarily recorded in the page tables. If an access is recorded, then it is treated like a load from the addressed byte. Some implementations may not take any action based on the execution of these instructions, or they may prefetch the cache block corresponding to the EA into their cache. For information about the R and C bits, see Section 7.6.3, “Page History Recording.”

Both **dcbt** and **dcbtst** are provided for performance optimization. These instructions do not affect the correct execution of a program, regardless of whether they succeed (fetch the cache block) or fail (do not fetch the cache block). If the target block is not accessible to the program for loads, then no operation occurs.

5.2.5.1.2 Data Cache Block Set to Zero (dcbz) Instruction

The **dcbz** instruction clears a single cache block as follows:

- If the target is in the data cache, all bytes of the cache block are cleared.
- If the target is not in the data cache and the corresponding page is caching-allowed, the cache block is established in the data cache (without fetching the cache block from main memory), and all bytes of the cache block are cleared.
- If the target is designated as either caching-inhibited or write-through, then either all bytes in main memory that correspond to the addressed cache block are cleared, or the alignment interrupt handler is invoked. The interrupt handler should clear all the bytes in main memory that correspond to the addressed cache block.
- If the target is designated as coherency required, and the cache block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches.

The **dcbz** instruction is treated as a store to the addressed byte with respect to address translation, protection, reference and change recording, and the ordering enforced by **eieio** or by the combination of caching-inhibited and guarded attributes for a page.

Section 6.5.2, “Machine Check Interrupt (0x00200),” describes a possible delayed machine check interrupt that can occur by using **dcbz** when the operating system has set up an incorrect memory mapping.

5.2.5.1.3 Data Cache Block Store (dcbst) Instruction

The **dcbst** instruction permits the program to ensure that the latest version of the target cache block is in main memory. The **dcbst** instruction executes as follows:

- Coherency required—If the target exists in the data cache(s) of any processor(s) and has been modified, the data is written to main memory.
- Coherency not required—If the target exists in the data cache of the executing processor and has been modified, the data is written to main memory.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target.

The memory access caused by **dcbst** is not necessarily recorded in the page tables. If the access is recorded, then it is treated as a load operation (not as a store operation).

5.2.5.1.4 Data Cache Block Flush (dcbf) Instruction

The action taken depends on the memory/cache access mode associated with the addressed byte and on the state of the cache block. The following list describes the action taken for the various cases:

- Coherency required

  Unmodified cache block—Invalidates copies of the cache block in the data caches of all processors.

  Modified cache block—Copies the cache block to memory. Invalidates copies of the cache block in the data caches of all processors.

  Target block not in cache—If a modified copy of the cache block is in any data cache of any processor, **dcbf** causes the modified cache block to be copied to memory and then invalidated. If unmodified copies are in the data caches of other processors, **dcbf** causes those copies to be invalidated.
- Coherency not required

  Unmodified cache block—Invalidates the cache block in the executing processor's data cache.

  Modified cache block—Copies the data cache block to memory and then invalidates the cache block in the executing processor.

  Target block not in cache—No action is taken.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target.

The memory access caused by **dcbf** is not necessarily recorded in the page tables. If the access is recorded, then it is treated as a load operation (not as a store operation).

5.2.5.2 Instruction Cache Instructions

Instruction caches, if they exist, are not required to be consistent with data caches, memory, or I/O data transfers. Software must use the appropriate cache management instructions to ensure that instruction caches are kept coherent when instructions are modified by the processor or by input data transfer. When a processor alters a memory location that may be contained in an instruction cache, software must ensure that updates to memory are visible to the instruction fetching mechanism. Although the instructions to enforce consistency vary among implementations, the following sequence for a uniprocessor system is typical:

1. **dcbst** (update memory)
2. **sync** (wait for update)
3. **icbi** (invalidate copy in instruction cache)
4. **isync** (perform context synchronization)

Note that most operating systems will provide a system service for this function. These operations are necessary because the memory may be designated as write-back. Because instruction fetching may bypass the data cache, changes made to items in the data cache may not otherwise be reflected in memory until after the fetch completes.

For implementations used in multiprocessor systems, variations on this sequence may be recommended. For example, in a multiprocessor system with a unified cache (at any level), if instructions are fetched without coherency being enforced, the preceding instruction sequence is inadequate. Because **icbi** does not invalidate blocks in a unified cache, a **dcbf** should be used instead of a **dcbst** for this case.

5.2.5.2.1 Instruction Cache Block Invalidate Instruction (icbi)

The **icbi** instruction executes as follows:

- Coherency required

  If the target is in the instruction cache of any processor, the cache block is made invalid in all such processors, so that the next reference causes the cache block to be refetched.
- Coherency not required

  If the target is in the instruction cache of the executing processor, the cache block is made invalid in the executing processor so that the next reference causes the cache block to be refetched.

The **icbi** instruction is provided for use in processors with separate instruction and data caches. The effective address is computed, translated, and checked for protection violations as defined in Section 7.5.4, “Block Memory Protection.” If the target block is not accessible to the program for loads, then a DSI interrupt occurs.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target.

A memory access caused by an **icbi** is not necessarily recorded in the page tables. If it is recorded, it is treated as a load operation. Implementations that have a unified cache treat **icbi** as a no-op except that they may invalidate the target cache block in the instruction caches of other processors (coherency is required).

5.2.5.2.2 Instruction Synchronize Instruction (isync)

The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** ensures that all instructions preceding the **isync** have completed before the **isync** completes, except that memory accesses caused by those instructions need not have been performed with respect to other processors and mechanisms. It also ensures that no subsequent instructions are initiated by the processor until after the **isync** completes. Finally, it causes the processor to discard any prefetched instructions, with the effect that subsequent instructions are fetched and executed in the context established by the instructions preceding **isync**. The **isync** has no effect on other processors or on their caches.

5.2.6 Shared Memory

The architecture supports sharing memory among programs, among different instances of the same program, and among processors and other mechanisms. It also supports access to a memory location by one or more programs using different effective addresses. In these cases, memory is shared in blocks that are an integral number of pages. When one physical memory location has different effective addresses, the addresses are said to be aliases. Each application can be granted separate access privileges to aliased pages.

Section 5.2.6.2, “Lock Acquisition and Import Barriers,” gives examples of how **sync** and **eieio** are used to control memory access ordering when memory is shared among programs.

5.2.6.1 Memory Access Ordering

The memory model defines memory access ordering as weakly consistent. This provides an opportunity for improved performance over a model with stronger consistency rules, but places the responsibility on the program to ensure that ordering or synchronization instructions are properly placed for the correct execution of the program.

The order in which the processor accesses memory, the order in which those accesses are performed with respect to other processors or mechanisms, and the order in which they are performed in main memory may all be different. The following ways to enforcing accesses ordering are provided to allow programs to share memory with other programs or with mechanisms such as I/O devices.

- If two store instructions specify memory locations that are both caching inhibited and guarded, the corresponding memory accesses are performed in program order with respect to any processor or mechanism.
- If a load instruction depends on the value returned by a preceding load (because the value is used to compute the effective address specified by the second load), the corresponding memory accesses are performed in program order with respect to any processor or mechanism to the extent required by the associated memory coherence required attributes (that is, the memory coherence required attribute, if any, associated with each access). This applies even if the dependency does not affect program logic (for example, the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load).
- When a processor (P1) executes **sync** or **eieio**, a memory barrier is created that separates applicable memory accesses into two groups, G1 and G2. G1 includes all applicable memory accesses

associated with instructions preceding the barrier-creating instruction, and G2 includes all applicable memory accesses associated with instructions following the barrier-creating instruction.

Figure 5-1 shows an example using a two-processor system.

<table>
<tr><th>Processor 1 (P1)</th><th>Memory Access Groups G1 and G2</th><th>Processor 2 (P2)</th></tr>
<tr><td>Instruction 1</td><td rowspan="4">G1: Memory accesses generated by P1 before the memory barrier</td><td rowspan="4">When memory coherence is required, G1 accesses that affect P2 are also performed before the memory barrier.</td></tr>
<tr><td>Instruction 2</td></tr>
<tr><td>Instruction 3</td></tr>
<tr><td>Instruction 4</td></tr>
<tr><td colspan="2">Instruction 5 (sync or eieio)—Memory barrier</td><td>Barrier generated by P1 does not order P2 instructions or associated accesses with respect to other P2 instructions and associated accesses.</td></tr>
<tr><td>Instruction 6</td><td rowspan="5">G2: Memory accesses generated by P1 after the memory barrier</td><td rowspan="5">When memory coherence is required, G2 accesses that affect P2 are also performed after the memory barrier.</td></tr>
<tr><td>Instruction 7</td></tr>
<tr><td>Instruction 8</td></tr>
<tr><td>Instruction 9</td></tr>
<tr><td>Instruction 10</td></tr>
</table>

Figure 5-1. Memory Barrier when Coherency is Required (M = 1)

The memory barrier ensures that all memory accesses in G1 are performed with respect to any processor or mechanism, to the extent required by the associated memory coherence required attributes (that is, the memory coherence required attribute, if any, associated with each access), before any memory accesses in G2 are performed with respect to that processor or mechanism.

The ordering done by a memory barrier is said to be cumulative if it also orders memory accesses that are performed by processors and mechanisms other than P1, as follows:

- G1 includes all applicable memory accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
- G2 includes all applicable memory accesses by any such processor or mechanism that are performed after a load instruction executed by that processor or mechanism has returned the value stored by a store that is in G2.

Figure 5-2 shows an example of a cumulative memory barrier in a two-processor system.

<table>
<tr><th>Processor 1 (P1)</th><th>Memory Access Groups G1 and G2</th><th>Processor 2 (P2)</th></tr>
<tr><td>P1 Instruction 1</td><td rowspan="4">G1: Memory accesses generated by P1 and P2 that affect P1. Includes accesses generated by executing P2 instructions L–O (assuming that the access generated by instruction O occurs before P1's sync is executed).</td><td>P2 Instruction L</td></tr>
<tr><td>P1 Instruction 2</td><td>P2 Instruction M</td></tr>
<tr><td>P1 Instruction 3</td><td>P2 Instruction N</td></tr>
<tr><td>P1 Instruction 4</td><td>P2 Instruction O</td></tr>
<tr><td colspan="2" rowspan="3">P1 Instruction 5 (sync)—Cumulative memory barrier applies to all accesses except those associated with fetching instructions following sync.</td><td>P2 Instruction P</td></tr>
<tr><td>P2 Instruction Q</td></tr>
<tr><td>P2 Instruction R</td></tr>
<tr><td>P1 Instruction 6</td><td rowspan="6">G2: Memory accesses generated by P1 and P2. Includes accesses generated by P2 instructions P–X (assuming that the access generated by instruction P occurs after P1's sync is executed) performed after a load instruction executed by P2 has returned the value stored by a store that is in G2.
The sync memory barrier does not affect accesses associated with instruction fetching that occur after the sync.</td><td>P2 Instruction S</td></tr>
<tr><td>P1 Instruction 7</td><td>P2 Instruction T</td></tr>
<tr><td>P1 Instruction 8</td><td>P2 Instruction U</td></tr>
<tr><td>P1 Instruction 9</td><td>P2 Instruction V</td></tr>
<tr><td>P1 Instruction 10</td><td>P2 Instruction W</td></tr>
<tr><td>P1 Instruction 11</td><td>P2 Instruction X</td></tr>
</table>

Figure 5-2. Cumulative Memory Barrier

A memory barrier created by **sync** is cumulative and applies to all accesses except those associated with fetching instructions following the **sync**. See the definition of **eieio** in Chapter 8, "Instruction Set," for a description of the corresponding properties of the memory barrier created by that instruction.

5.2.6.1.1 Programming Considerations

Because stores cannot be performed out of program order, as described in the OEA, if a store instruction depends on the value returned by a preceding load (because the value the load returns is needed to compute either the effective address specified by the store or the value to be stored), the corresponding accesses are performed in program order. The same applies if whether the store instruction executes depends on a conditional branch that in turn depends on the value returned by a preceding load. For example, if a conditional branch depends on a preceding load and that branch chooses between a path that includes a store instruction if the condition is met, that dependent store is not performed unless and until the condition determined by the load is met.

Because instructions following an **isync** cannot execute until all instructions preceding it have completed, if an **isync** follows a conditional branch instruction that depends on the value returned by a preceding load instruction, that load is performed before any loads caused by instructions following the **isync**. This is true even if the effects of the dependency are independent of the value loaded (for example, the value is compared to itself and the branch tests CR*n*[EQ]), and even if the branch target is the next sequential instruction.

Except for the cases described above and earlier in this section, data and control dependencies do not order memory accesses. Examples include the following:

- If a load specifies the same memory location as a preceding store and the location is not caching inhibited, the load may be satisfied from a store queue (a buffer into which the processor places stored values before presenting them to the memory subsystem) and not be visible to other processors and mechanisms. As a result, if a subsequent store depends on the value returned by the load, the two stores need not be performed in program order with respect to other processors and mechanisms.
- Because a store conditional instruction may complete before its store is performed, a conditional branch instruction that depends on the CR0 value set by a store conditional instruction does not order the store conditional's store with respect to memory accesses caused by instructions that follow the branch.

  For example, in the following sequence, the **stw** instruction is the **bc** instruction's target:

  stwcx.
  bc
  stw

  For the **stwcx.** to complete, it must update the architected CR0 value, even though its store may not have performed. The architecture does not require that the store generated by the **stwcx.** must be performed before the store generated by the **stw**.
- Because processors may predict branch target addresses and branch condition resolution, control dependencies (for example, branches) do not order memory accesses except as described above. For example, when a subroutine returns to its caller, the return address may be predicted, with the result that loads caused by instructions at or after the return address may be performed before the load that obtains the return address is performed.

Some processors implement nonarchitected duplicates of architected resources such as GPRs, CR fields, and the LR, so resource dependencies (for example, specification of the same target register for two load instructions) do not order memory accesses.

Examples of correct uses of dependencies, **sync**, and **eieio** to order memory accesses can be found in Appendix D, "Synchronization Programming Examples."

Because the memory model is weakly consistent, the sequential execution model as applied to instructions that cause memory accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions. For example, an instruction may complete, and subsequent instructions may be executed, before memory accesses caused by the first instruction have been performed. However, for a sequence of atomic accesses to the same memory location for which memory coherence is required, the definition of coherence guarantees that the accesses are performed in program order with respect to any processor or mechanism that accesses the location coherently, and similarly if the location is one for which caching is inhibited.

Because accesses to caching inhibited memory are performed in main memory, memory barriers and dependencies on load instructions order such accesses with respect to any processor or mechanism even if the memory is not memory coherence required.

5.2.6.1.2 Programming Examples

Example 1 shows cumulative ordering of memory accesses preceding a memory barrier, and the second illustrates cumulative ordering of memory accesses following a memory barrier. Assume that locations X, Y, and Z initially contain the value 0.

Example 1:

- Processor A stores the value 1 to location X.
- Processor B loads from location X obtaining the value 1, executes a **sync**, then stores the value 2 to location Y.
- Processor C loads the value 2 from location Y, executes a **sync**, then loads from location X.

Example 2:

- Processor A stores the value 1 to location X, executes a **sync**, then stores the value 2 to location Y.
- Processor B loops loading from location Y until the value 2 is obtained, then stores the value 3 to location Z.
- Processor C loads from location Z obtaining the value 3, executes a **sync**, then loads from location X. In both cases, cumulative ordering dictates that the value loaded from location X by processor C is 1.

5.2.6.2 Lock Acquisition and Import Barriers

An import barrier is an instruction or instruction sequence that prevents memory accesses caused by instructions following the barrier from being performed before memory accesses that acquire a lock have been performed. An import barrier can be used to ensure that a shared data structure protected by a lock is not accessed until the lock has been acquired. A **sync** instruction can always be used as an import barrier, but the approaches shown below generally yield better performance because they order only the relevant memory accesses.

5.2.6.2.1 Acquire Lock and Import Shared Memory

If **lwarx** and **stwcx.** are used to obtain the lock, an import barrier can be constructed by placing an **isync** immediately following the loop containing the **lwarx** and **stwcx.**. The following example uses the Compare and Swap primitive (see Section D.2, “Synchronization Primitives”) to acquire the lock.

This example assumes that the address of the lock is in GPR 3, the value indicating that the lock is free is in GPR 4, the value to which the lock should be set is in GPR 5, the old value of the lock is returned in GPR 6, and the address of the shared data structure is in GPR 9.

```
loop:    lwarx    r6,0,r3          # load lock and reserve
         cmpw     r4,r6            # skip ahead if
         bne-     wait             # lock not free
         stwcx.   r5,0,r3          # try to set lock
         bne-     loop             # loop if lost reservation
         isync                     # import barrier
         lwz      r7,data1(r9)     # load shared data
.
.
wait: ...                          #wait for lock to free
```

The second **bne-** does not complete until CR0 has been set by the **stwcx.**. The **bne-** simplified mnemonic indicates that the branch is predicted as not taken. See Appendix E, “Simplified Mnemonics for PowerPC Instructions.” The **stwcx.** does not set CR0 until it has completed (successfully or unsuccessfully). The lock is acquired when the **stwcx.** completes successfully. Together, the second **bne-** and the subsequent **isync** create an import barrier that prevents the load from data1 from being performed until the branch has been resolved not to be taken.

5.2.6.2.2 Obtain Pointer and Import Shared Memory

If **lwarx** and **stwcx.** are used to obtain a pointer into a shared data structure, an import barrier is not needed if all the accesses to the shared data structure depend on the value obtained for the pointer. The following example uses the Fetch and Add primitive (see Section D.2, “Synchronization Primitives”) to obtain and increment the pointer.

In this example it is assumed that the address of the pointer is in GPR 3, the value to be added to the pointer is in GPR 4, and the old value of the pointer is returned in GPR 5.

```
loop:    lwarx    r5,0,r3          # load pointer and reserve
         add      r0,r4,r5         # increment the pointer
         stwcx.   r0,0,r3          # try to store new value
         bne-     loop             # loop if lost reservation
         lwz      r7,data1(r5)     # load shared data
```

The load from data1 cannot be performed until the **lwarx** loads the pointer value into GPR 5. The load from data1 may be performed out-of-order before the **stwcx.**. But if the **stwcx.** fails, the branch is taken and the value returned by the load from data1 is discarded. If the **stwcx.** succeeds, the value returned by the load from data1 is valid even if the load is performed out-of-order, because the load uses the pointer value returned by the instance of the **lwarx** that created the reservation used by the successful **stwcx.**.

An **isync** could be placed between the **bne-** and the subsequent **lwz**, but no **isync** is needed if all accesses to the shared data structure depend on the value returned by the **lwarx**.

5.3 The Operating Environment

The OEA defines the mechanism for controlling the memory/cache access modes introduced in Section 5.2.4.1, “Memory/Cache Access Modes.” This section describes the cache-related aspects of the OEA including the memory/cache access attributes, the **dcbi**, and speculative execution. Note that the terms ‘speculative’ and ‘out-of-order’ are used here as defined in Section 5.3.1.5.1, “Definition of Speculative and Out-of-Order Memory Accesses.”

The features of the OEA are accessible to supervisor-level applications only. The mechanism for controlling the virtual memory space is described in Chapter 7, “Memory Management.”

The memory model of PowerPC processors provides the following features:

- Flexibility to allow performance benefits of weakly-ordered memory access
- A mechanism to maintain memory coherency among processors and between a processor and I/O devices controlled at the block and page level
- Instructions that can be used to ensure a consistent memory state
- Guaranteed processor access order

The memory implementations in PowerPC systems can take advantage of the performance benefits of weak ordering of memory accesses between processors or between processors and other external devices without any additional complications. Memory coherency can be enforced externally by a snooping bus design, a centralized cache directory design, or other designs that can take advantage of the coherency features of PowerPC processors.

Memory accesses performed by a single processor appear to complete sequentially from the view of the programming model but may complete out of order with respect to the ultimate destination in the memory hierarchy. Order is guaranteed at each level of the memory hierarchy for accesses to the same address from the same processor. The **dcbst**, **dcbf**, **icbi**, **isync**, **sync**, **eieio**, **lwarx**, and **stwcx.** instructions allow the programmer to ensure a consistent memory state.

5.3.1 Memory/Cache Access Attributes

All instruction and data accesses are performed under the control of the four memory/cache access attributes:

- Write-through (W attribute)
- Caching-inhibited (I attribute)
- Memory coherency (M attribute)
- Guarded (G attribute)

The operating system programs these attributes in the PTEs and BATs for each page and block respectively. The W and I attributes control how the processor performing an access uses its own cache. The M attribute ensures that coherency is maintained for all copies of the addressed memory location. When an access requires coherency, the processor performing the access must inform the coherency mechanisms throughout the system that the access requires memory coherency. The G attribute prevents speculative (referred to as out-of-order in the architecture specification) loading and prefetching from the addressed memory location.

Note that the memory/cache access attributes are relevant only when an effective address is translated by the processor performing the access. Note also that not all combinations of settings of these bits is supported. The attributes are not saved along with data in the cache (for cacheable accesses), nor are they associated with subsequent accesses made by other processors.

The operating system programs the memory/cache access attribute for each page or block as required. The WIMG attributes occupy four bits in the BAT registers for block address translation and in the PTEs for page address translation. The WIMG bits are programmed as follows:

- The operating system uses **mtspr** to program the WIMG bits in the BAT registers for block address translation. The IBAT register pairs implement the W or G bits; however, attempting to set either bit in IBAT registers causes boundedly-undefined results.
- The operating system writes the WIMG bits for each page into the PTEs in system memory as it sets up the page tables.

Note that for data accesses performed in real addressing mode (MSR[DR] = 0), the WIMG bits are assumed to be 0b0011 (the data is write-back, caching is enabled, memory coherency is enforced, and memory is guarded). For instruction accesses performed in real addressing mode (MSR[IR] = 0), the

WIMG bits are assumed to be 0b0001 (the data is write-back, caching is enabled, memory coherency is not enforced, and memory is guarded).

5.3.1.1 Write-Through Attribute (W)

When an access is designated as write-through (W = 1), if the data is in the cache, a store operation updates the cached copy of the data. In addition, the update is written to the memory location. The definition of the memory location to be written to (in addition to the cache) depends on the implementation of the memory system but can be illustrated by the following examples:

- RAM—The store is sent to the RAM controller to be written into the target RAM.
- I/O device—The store is sent to the memory-mapped I/O controller to be written to the target register or memory location.

In systems with multilevel caching, the store must be written to at least a depth in the memory hierarchy that is seen by all processors and devices.

Multiple store instructions may be combined for write-through accesses except when the store instructions are separated by a **sync** or **eieio**. A store operation to a memory location designated as write-through may cause any part of the cache block to be written back to main memory.

Accesses that correspond to W = 0 are considered write-back. For this case, although the store operation is performed to the cache, the data is copied to memory only when a copy-back operation is required. Use of the write-back mode (W = 0) can improve overall performance for areas of the memory space that are seldom referenced by other processors or devices in the system.

Accesses to the same memory location using two effective addresses for which the W bit setting differs meet the memory-coherency requirements if the accesses are performed by a single processor. If the accesses are performed by two or more processors, coherence is enforced by the hardware only if the write-through attribute is the same for all of the accesses.

5.3.1.2 Caching-Inhibited Attribute (I)

If I = 1, the memory access is completed by referencing the location in main memory, bypassing the cache. During the access, the addressed location is not loaded into the cache nor is the location allocated in the cache.

It is considered a programming error if a copy of the target location of an access to caching-inhibited memory is resident in the cache. Software must ensure that the location has not been previously loaded into the cache, or, if it has, that it has been flushed from the cache.

Data accesses from more than one instruction may be combined for cache-inhibited operations, except when the accesses are separated by a **sync**, or by an **eieio** when the page or block is also designated as guarded.

Instruction fetches, **dcbz** instructions, and load and store operations to the same memory location using two effective addresses for which the I bit setting differs must meet the requirement that a copy of the target location of an access to caching-inhibited memory not be in the cache. Violation of this requirement is considered a programming error; software must ensure that the location has not previously been brought into the cache or, if it has, that it has been flushed from the cache. If the programming error occurs, the

result of the access is boundedly undefined. It is not considered a programming error if the target location of any other cache management instruction to caching-inhibited memory is in the cache.

5.3.1.3 Memory Coherency Attribute (M)

The memory coherency attribute is provided to allow improved performance in systems where hardware-enforced coherency is relatively slow, and software is able to enforce the required coherency. When M = 0, there are no requirements to enforce data coherency. When M = 1, the processor enforces data coherency.

When M is set and the access is performed to memory, hardware indicates to the rest of the system that the access is global. Other processors affected by this global access must then respond to it. For example, in a snooping bus design, the processor may assert some type of global access signal. Other processors affected by the access respond and signal whether the data is being shared. If the data in another processor is modified, the location is updated and the access is retried.

Because instruction memory does not have to be coherent with data memory, some implementations may ignore the M attribute for instruction accesses. In a single-processor (or single-cache) system, performance might be improved by designating all pages as memory coherency not required.

Accesses to the same memory location using two effective addresses for which the M bit settings differ may require explicit software synchronization before accessing the location with M = 1 if the location has previously been accessed with M = 0. Any such requirement is system-dependent. For example, no software synchronization may be required for systems that use bus snooping. In some directory-based systems, software may be required to execute **dcbf** instructions on each processor to flush all storage locations accessed with M = 0 before accessing those locations with M = 1.

5.3.1.4 W, I, and M Bit Combinations

Table 5-1 summarizes the six combinations of the WIM bits supported by the OEA. The combinations where WIM = 11x are not supported. Note that either a zero or one setting for the G bit is allowed for each of these WIM bit combinations.

Table 5-1. Combinations of W, I, and M Bits

WIM	Meaning
000	The processor may cache data (or instructions). A load or store operation whose target hits in the cache can use that entry in the cache. The processor does not need to enforce memory coherency for accesses it initiates.
001	Data (or instructions) may be cached. A load or store operation whose target hits in the cache may use that entry in the cache. The processor enforces memory coherency for accesses it initiates.
010	Caching is inhibited. The access is performed to memory, bypassing the cache. The processor does not need to enforce memory coherency for accesses it initiates.
011	Caching is inhibited. The access is performed to memory, bypassing the cache. The processor enforces memory coherency for accesses it initiates.

Table 5-1. Combinations of W, I, and M Bits (continued)

WIM	Meaning
100	Data (or instructions) may be cached. A load operation whose target hits in the cache may use that entry in the cache. Store operations are written to memory. The target location of the store may be cached and is updated on a hit. The processor does not need to enforce memory coherency for accesses it initiates.
101	Data (or instructions) may be cached. A load operation whose target hits in the cache may use that entry in the cache. Store operations are written to memory. The target location of the store may be cached and is updated on a hit. The processor enforces memory coherency for accesses it initiates.

5.3.1.5 The Guarded Attribute (G)

When the guarded bit is set, the memory area (block or page) is designated as guarded. This setting is used to protect certain memory areas from read accesses made by the processor that are not dictated directly by the program. If there are areas of physical memory that are not fully populated (in other words, there are holes in the physical memory map within this area), this setting protects the system from undesired accesses caused by speculative (referred to as out-of-order in the architecture specification) load operations or instruction prefetches that could lead to the generation of the machine check interrupt. Also, the guarded bit is used to prevent speculative load operations or prefetches from occurring to certain peripheral devices that produce undesired results when accessed in this way.

5.3.1.5.1 Definition of Speculative and Out-of-Order Memory Accesses

In the architecture definition, the term ‘out-of-order’ replaced the term ‘speculative’ with respect to memory accesses to avoid a conflict between the word’s meaning in the context of execution of instructions past unresolved branches. The architecture’s use of out-of-order in this context could in turn be confused with the notion of loads and stores being reordered in a weakly ordered memory system.

To address the need for these distinctions, in the context of memory accesses this document uses the terms ‘speculative’ and ‘out-of-order’ as follows:

- Speculative memory access. An access to memory that occurs before it is known to be required by the sequential execution model.
- Out-of-order memory access. A memory access performed ahead of one that may have preceded it in the sequential model, such as is allowed by a weakly-ordered memory model.

5.3.1.5.2 Performing Operations Speculatively

An operation is said to be nonspeculative if it is guaranteed to be required by the sequential execution model. Any other operation is said to be performed speculatively, which the architecture specification refers to as out of order.

Operations are performed speculatively by hardware on the expectation that the results will be needed by an instruction that will be required by the sequential execution model. Whether the results are needed depends on anything that might divert the program flow away from the instruction, such as interrupts, branch, trap, system call, and **rfi** instructions, and anything that might change the context in which the instruction is executed.

Typically, the hardware performs operations speculatively when it has resources that would otherwise be idle, so the operation incurs little or no cost. If subsequent events such as branches or interrupts indicate that the operation would not have been performed in the sequential execution model, the processor abandons any results of the operation (except as described below).

Most operations can be performed speculatively, as long as the processor appears to follow the sequential execution model. Certain speculative operations are restricted, as follows.

- Stores—A store instruction may not be executed speculatively in a manner such that the alteration of the target location can be observed by other processors or mechanisms.
- Accessing guarded memory—The restrictions for this case are given in Section 5.3.1.5.4, “Speculative Accesses to Guarded Memory.”

Only a machine check interrupt may be reported due to an operation that is performed speculatively, until such time as it is known that the operation is required by the sequential execution model. The only other permitted side effects (other than machine check) of performing an operation speculatively are the following:

- Reference and change bits may be set as described in Section 7.6.3, “Page History Recording.”
- Nonguarded memory locations that could be fetched into a cache by nonspeculative execution may be fetched speculatively into that cache.

5.3.1.5.3 Guarded Memory

Memory is said to be well behaved if the corresponding physical memory exists and is not defective and if the effects of a single access to it are indistinguishable from the effects of multiple identical accesses to it. Data and instructions can be fetched speculatively from well-behaved memory without causing undesired side effects.

Memory is said to be guarded if either of the following cases:

- The G bit is 1 in the relevant PTE or DBAT register
- The processor is in real addressing mode (MSR[IR] = 0 or MSR[DR] = 0 for instruction fetches or data accesses respectively). In this case, all of memory is guarded for the corresponding accesses.

In general, memory that is not well-behaved should be guarded. Because such memory may represent an I/O device or may include non-existent locations, a speculative access to such memory may cause an I/O device to perform incorrect operations or may cause a machine check.

Note that if separate store instructions access memory that is both caching-inhibited and guarded, the accesses are performed in the order specified by the program. If an aligned, load or store that is not a string or multiple access to caching-inhibited, guarded memory has accessed main memory and an external, decrementer, or imprecise-mode floating-point enabled interrupt is pending, the load or store is completed before the interrupt is taken.

5.3.1.5.4 Speculative Accesses to Guarded Memory

The circumstances in which guarded memory may be accessed speculatively are as follows:

- Load instruction. If a copy of the target location is in a cache, the location may be accessed in the cache or in main memory.
- Instruction fetch. In real addressing mode (MSR[IR] = 0), an instruction may be fetched if any of the following conditions is met:
  - The instruction is in a cache. In this case, it may be fetched from that cache.
  - The instruction is in the same physical page as an instruction that is required by the sequential execution model or is in the physical page immediately following such a page.

  If MSR[IR] = 1, instructions may not be fetched from either no-execute segments or guarded memory. If the effective address of the current instruction is mapped to either of these kinds of memory when MSR[IR] = 1, an instruction storage interrupt is generated. However, it is permissible for an instruction from either of these kinds of memory to be in the instruction cache if it was fetched into that cache when its effective address was mapped to some other kind of memory. Thus, for example, the operating system can access an application's instruction segments as no-execute without having to invalidate them in the instruction cache.

  Additionally, instructions are not fetched from direct-store segments (only applies when MSR[IR] = 1). If an instruction fetch is attempted from a direct-store segment, an instruction storage interrupt is generated. Note that the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

Note that software should ensure that only well-behaved memory is loaded into a cache, either by marking as caching-inhibited (and guarded) all memory that may not be well-behaved, or by marking such memory caching-allowed (and guarded) and referring only to cache blocks that are well-behaved.

If a physical page contains instructions to be executed in real addressing mode (MSR[IR] = 0), software should ensure that this and the next physical page contain only well-behaved memory.

5.3.2 I/O Interface Considerations

The PowerPC architecture defines two mechanisms for accessing I/O:

- Memory-mapped I/O interface operations. SR[T] = 0. These operations are considered to address memory space and are therefore subject to the same coherency control as memory accesses. Depending on the specific I/O interface, the memory/cache access attributes (WIMG) and the degree of access ordering (requiring **eieio** or **sync** instructions) need to be considered. This is the recommended way of accessing I/O.
- Direct-store segment operations. SR[T] = 1. These operations are considered to address the noncoherent and noncacheable direct-store segment space; therefore, hardware need not maintain coherency for these operations, and the cache is bypassed completely. Although the architecture defines this direct-store functionality, it is being phased out of the architecture and will not likely be supported in future devices. Thus, its use is discouraged, and new software should not use it or depend on its effects.

5.3.3 OEA Cache Management Instruction—Data Cache Block Invalidate (dcbi)

As described in Section 5.2.5, “VEA Cache Management Instructions,” the VEA defines instructions for controlling both the instruction and data caches, The OEA defines one instruction, the data cache block invalidate (**dcbi**) instruction, for controlling the data cache. This section briefly describes the cache management instruction available to programs at the supervisor privilege level. Additional descriptions of coding **dcbi** are provided in Section 4.4.3.1, “Supervisor-Level Cache Management Instruction,” and Chapter 8, “Instruction Set.” In the following description, the target is the cache block containing the byte addressed by the effective address.

Any cache management instruction that generates an EA that corresponds to a direct-store segment (SR[T] = 1) is treated as a no-op. However, note that the direct-store facility is being phased out of the architecture and will not likely be supported in future devices. Thus, software should not depend on its effects.

The action taken depends on the memory/cache access mode associated with the target, and on the state of the cache block. The following list describes the action taken for the various cases:

- Coherency required

  Unmodified cache block—Invalidates copies of the cache block in the data caches of all processors.

  Modified cache block—Invalidates copies of the cache block in the data caches of all processors. (Discards the modified data in the cache block.)

  Target block not in cache—If copies of the target are in the data caches of other processors, **dcbi** causes those copies to be invalidated, regardless of whether the data is modified or unmodified.
- Coherency not required

  Unmodified cache block—Invalidates the cache block in the executing processor's data cache.

  Modified cache block—Invalidates the cache block in the executing processor's data cache. (Discards the modified data in the cache block.)

  Target block not in cache—No action is taken.

The processor treats **dcbi** as a store to the addressed byte with respect to address translation and protection. It is not necessary to set the reference and change bits.

The function of this instruction is independent of the write-through/write-back and caching-inhibited/caching-allowed attributes of the target. To ensure coherency, aliased effective addresses (two effective addresses that map to the same physical address) must have the same page offset.

Chapter 6
Interrupts

This chapter describes the interrupt model defined in the OEA. The following topics are covered:

- Interrupt Classes
- Interrupt Processing
- Process Switching
- Interrupt Definitions

6.1 Overview

The operating environment architecture (OEA) portion of the PowerPC architecture defines the mechanism by which PowerPC processors implement interrupts. Interrupt conditions that cause interrupts may be defined at other levels of the architecture. For example, the user instruction set architecture (UISA) defines conditions that may cause floating-point interrupts; the OEA defines the mechanism by which the interrupt is taken.

The PowerPC interrupt mechanism allows the processor to change to supervisor state as a result of external signals, errors, or unusual conditions arising in the execution of instructions. When interrupts occur, information about the state of the processor is saved to certain registers and the processor begins execution at an address (interrupt vector) predetermined for each interrupt. Processing of interrupts begins in supervisor mode.

Although multiple interrupt conditions can map to a single interrupt vector, a more specific condition may be determined by examining a register associated with the interrupt—for example, the DSISR and the floating-point status and control register (FPSCR). Additionally, certain interrupt conditions and interrupts can be explicitly enabled or disabled by software.

The PowerPC architecture requires that interrupts be taken in program order; therefore, although a particular implementation may recognize interrupt conditions out of order, the interrupts associated with them are handled strictly in order with respect to the instruction stream. When an instruction-caused interrupt is recognized, any unexecuted instructions that appear earlier in the instruction stream, including any that have not yet entered the execute state, are required to complete before the interrupt is taken. For example, if a single instruction encounters multiple interrupt conditions, those interrupts are taken and handled sequentially. Likewise, asynchronous, precise interrupts are recognized when they occur, but are not handled until all instructions currently in the execute stage successfully complete execution and report their results.

Note that interrupts can occur while an interrupt handler routine is executing and that multiple interrupts can become nested. It is up to the interrupt handler to save the appropriate machine state if it is desired to allow control to ultimately return to the excepting program.

In many cases, after the interrupt handler handles an interrupt, there is an attempt to execute the instruction that caused the interrupt. Instruction execution continues until the next interrupt condition is encountered. This method of recognizing and handling interrupts sequentially guarantees that the machine state is recoverable and processing can resume without losing instruction results.

To prevent the loss of state information, interrupt handlers must save the information stored in SRR0 and SRR1 soon after the interrupt is taken to prevent this information from being lost due to another interrupt being taken.

In this chapter, the following terminology is used to describe the various stages of interrupt processing:

Exception	The condition that can cause an interrupt. For some interrupts, such as the program interrupt, there are many exception conditions that can generate an interrupt.
Recognition	Recognition occurs when the exception condition that can cause an interrupt is identified by the processor.
Taken	An interrupt is said to be taken when control of instruction execution is passed to the interrupt handler; that is, the context is saved and the instruction at the appropriate vector offset is fetched and the interrupt handler routine is begun in supervisor mode.
Handling	Interrupt handling is performed by the software linked to the appropriate vector offset. Interrupt handling is begun in supervisor mode (referred to as privileged state in the architecture specification).

6.2 Interrupt Classes

As specified by the PowerPC architecture, all interrupts can be described as either precise or imprecise and either synchronous or asynchronous. Asynchronous interrupts are caused by events external to the processor's execution; synchronous interrupts are caused by instructions.

The PowerPC interrupt types are shown in Table 6-1.

Table 6-1. PowerPC Interrupt Classifications

Type	Interrupt
Asynchronous/nonmaskable	Machine check, system reset
Asynchronous/maskable	External interrupt, decrementer
Synchronous/precise	Instruction-caused interrupts, excluding floating-point imprecise interrupts
Synchronous/imprecise	Instruction-caused imprecise interrupts (floating-point imprecise interrupts)

Interrupts, their offsets, and exception conditions that cause them, are summarized in Table 6-2. The interrupt vectors described in the table correspond to physical address locations, depending on the value of MSR[IP]. Section 7.3.1.1, "Predefined Physical Memory Locations," lists all of the predefined physical memory areas. Remaining sections in this chapter provide more complete descriptions of the interrupts and exception conditions.

Table 6-2. Interrupts and Exception Conditions—Overview

Interrupt Type	Vector Offset (hex)	Causing Conditions
System reset	00100	Causes are implementation-dependent. If the conditions that cause the interrupt also cause the processor state to be corrupted such that the contents of SRR0 and SRR1 are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the RI bit copied from the MSR to SRR1 is cleared.
Machine check	00200	Causes are implementation-dependent, but typically are related to conditions such as bus parity errors or attempting to access an invalid physical address. Typically, these interrupts are triggered by an input signal to the processor. Note that not all processors provide the same level of error checking. The machine check interrupt is disabled when MSR[ME] = 0. If a machine check interrupt condition exists and ME = 0 is cleared, the processor goes into checkstop state. If the conditions that cause the interrupt also corrupt the processor state such that SRR0 and SRR1 contents are no longer valid or such that other processor resources are so corrupted that the processor cannot reliably resume execution, the copy of the MSR[RI] written from the MSR to SRR1 is cleared. Note that physical address is referred to as real address in the architecture specification.
Data Storage	00300	Occurs when a data memory access cannot be performed for any of the reasons described in Section 6.5.3, “Data Storage Interrupt (0x00300).” Such accesses can be generated by load/store instructions and by certain memory control and cache control instructions.
Instruction Storage	00400	Occurs when an instruction fetch cannot be performed for a variety of reasons described in Section 6.5.4, “Instruction Storage Interrupt (0x00400).”
External interrupt	00500	Generated only when an external interrupt is pending (typically signalled by a signal defined by the implementation) and the interrupt is enabled (MSR[EE] = 1).
Alignment	00600	May occur when the processor cannot perform a memory access for reasons described in Section 6.5.6, “Alignment Interrupt (0x00600).” Note that an implementation is allowed to perform the operation correctly and not cause an alignment interrupt.
Program	00700	Caused by one of the following exception conditions, which correspond to bit settings in SRR1 and arise during execution of an instruction: • Floating-point enabled exception—Generated when MSR[FE0,FE1] ≠ 00 and FPSCR[FEX] is set. Table 6-3 describes FE0 and FE1 settings. FPSCR[FEX] is set by execution of a floating-point instruction that causes an enabled exception or by the execution of a Move to FPSCR instruction that sets both an exception condition bit and its corresponding enable bit in the FPSCR. These interrupts are described in Section 3.3.6, “Floating-Point Program Exceptions.” • Illegal instruction—Generated by attempted execution of an instruction with an illegal opcode or illegal combination of opcode and extended opcode fields or when execution of an optional instruction not provided in the specific implementation is attempted (these do not include those optional instructions that are treated as no-ops). The instruction set is described in Chapter 4, “Addressing Modes and Instruction Set Summary.” See Section 6.5.7, “Program Interrupt (0x00700),” for a list of exceptions. • Privileged instruction—Generated by attempted execution of a privileged instruction when the MSR user privilege bit, MSR[PR], is set. Also generated for **mtspr** or **mfspr** with an invalid SPR field if SPR[0] = 1 and MSR[PR] = 1. • Trap—Generated when any of the conditions specified in a trap instruction is met. For more information, refer to Section 6.5.7, “Program Interrupt (0x00700).”
Floating-point unavailable	00800	Caused by an attempt to execute a floating-point instruction (including floating-point load, store, and move instructions) when the floating-point available bit is cleared, MSR[FP] = 0.

Table 6-2. Interrupts and Exception Conditions—Overview (continued)

Interrupt Type	Vector Offset (hex)	Causing Conditions
Decrementer	00900	Created when the msb of the decrementer changes from 0 to 1. Taken if the interrupt is enabled (MSR[EE] = 1). If it is not enabled, the interrupt remains pending until it is taken.
Reserved	00A00	Reserved for implementation-specific interrupts.
Reserved	00B00	—
System call	00C00	Occurs when a System Call (**sc**) instruction is executed.
Trace	00D00	Optional. If implemented, a trace interrupt occurs if either MSR[SE] = 1 and almost any instruction completes successfully or MSR[BE] = 1 and a branch instruction is completed. See Section 6.5.11, “Trace Interrupt (0x00D00).”
Floating-point assist	00E00	Optional. This interrupt can be used to provide software assistance for infrequent and complex floating-point operations such as denormalization.
Reserved	00E10–00FFF	—
Reserved	01000–02FFF	Reserved, may be used for implementation-specific interrupt vectors or other uses.

6.2.1 Precise Interrupts

When any precise interrupts occur, SRR0 is set to point to an instruction such that all prior instructions in the instruction stream have completed execution and no subsequent instruction has begun execution. However, depending on the interrupt type, the instruction addressed by SRR0 may not have completed execution.

When an interrupt occurs, instruction dispatch is halted and the following synchronization is performed:

1. The interrupt mechanism waits for all previous instructions in the instruction stream to complete to a point where they report all interrupts they will cause. However, some memory accesses associated with these preceding instructions may not have been performed with respect to other processors and mechanisms.
2. The processor ensures that all previous instructions in the instruction stream complete in the context in which they began execution.
3. The interrupt mechanism implemented in hardware and the software handler is responsible for saving and restoring the processor state.

The synchronization described conforms to the requirements for context synchronization. A complete description of context synchronization is described in the following section.

6.2.2 Context Synchronization

The synchronization described in Section 4.1.4, “Synchronizing Instructions,” can be extended to cover interrrupts as well. An instruction or event is context synchronizing if it satisfies all the requirements listed below. Context-synchronizing operations include the **sc** and **rfi** instructions and most interrupts and have the following characteristics:

- The operation causes instruction dispatching to be halted.

- The operation is not initiated or, in the case of **isync**, does not complete, until all instructions in execution have completed to a point at which they have reported all interrupts they will cause.
- Instructions that precede the operation complete execution in the context (for example, the privilege, translation mode, and memory protection) in which they were initiated.
- If the operation either directly causes an interrupt (for example, the **sc** instruction causes a system call interrupt) or is an interrupt, the operation is not initiated until no interrupt exists having higher priority than the interrupt associated with the context-synchronizing operation.

A context-synchronizing operation is necessarily execution synchronizing. Unlike the **sync** instruction, these operations need not wait for memory-related operations to complete on other processors or for reference and change bits in the page table to be updated.

Execution synchronization is described in Section 4.1.4.2, “Execution Synchronizing Instructions.”

6.2.2.1 Execution Synchronization

An instruction is execution synchronizing if it satisfies the conditions of the first two items described above for context synchronization. The **sync** instruction is treated like **isync** with respect to the second item described above (that is, the conditions described in the second item apply to the completion of **sync**). The **sync** and **mtmsr** instructions are examples of execution-synchronizing instructions.

All context-synchronizing instructions are execution-synchronizing. Unlike a context-synchronizing operation, an execution-synchronizing instruction need not ensure that the subsequent instructions execute in the context established by that instruction. This new context becomes effective sometime after the execution-synchronizing instruction completes and before or at a subsequent context-synchronizing operation.

6.2.2.2 Synchronous/Precise Interrupts

When instruction execution causes a precise interrupt, the following conditions exist at the interrupt point:

- Depending on the type of interrupt, SRR0 addresses either the instruction causing the interrupt or the immediately following instruction. The instruction addressed can be determined from the interrupt type and status bits, which are defined in the description of each interrupt.
- All instructions that precede the excepting instruction complete before the interrupt is processed. However, some memory accesses generated by these preceding instructions may not have been performed with respect to all other processors or system devices.
- The instruction causing the interrupt may not have begun execution, may have partially completed, or may have completed, depending on the interrupt type. Handling of partially executed instructions is described in Section 6.2.4, “Partially Executed Instructions.”
- Architecturally, no subsequent instruction has begun execution.

While instruction parallelism allows the possibility of multiple instructions reporting interrupts during the same cycle, they are handled one at a time in program order. Interrupt priorities are described in Section 6.2.5, “Interrupt Priorities.”

6.2.2.3 Asynchronous Interrupts

There are four asynchronous interrupts—system reset and machine check—which are nonmaskable and highest-priority interrupts, and external interrupt and decrementer interrupts, which are maskable and low-priority. These two types of asynchronous interrupts are discussed separately.

6.2.2.3.1 System Reset and Machine Check Interrupts

System reset and machine check interrupts have the highest priority and can occur while other interrupts are being processed. Note that nonmaskable, asynchronous interrupts are never delayed; therefore, if two of these interrupts occur in immediate succession, the state information saved by the first interrupt may be overwritten when the subsequent interrupt occurs. Note that these interrupts are context-synchronizing if they are recoverable (MSR[RI] is copied from the MSR to SRR1 if the interrupt does not cause loss of state.) If the RI bit is clear (nonrecoverable), the interrupt is context-synchronizing only with respect to subsequent instructions.

These interrupts cannot be masked by using MSR[EE]. However, if the machine check enable bit, MSR[ME], is cleared and a machine check exception condition occurs, the processor goes directly into checkstop state as the result of the exception. When one of these interrupts occurs, the following conditions exist at the interrupt point:

- For system reset interrupts, SRR0 addresses the instruction that would have attempted to execute next if the interrupt had not occurred.
- For machine check interrupts, SRR0 holds either an instruction that would have completed or some instruction following it that would have completed if the interrupt had not occurred.
- An interrupt is generated such that all instructions preceding the instruction addressed by SRR0 appear to have completed with respect to the executing processor.

Note that MSR[RI] indicates whether enough of the machine state was saved to allow processing to resume.

6.2.2.3.2 External and Decrementer Interrupts

For the external interrupt and decrementer interrupts, the following conditions exist at the interrupt point (assuming these interrupts are enabled (MSR[EE] is set)):

- All instructions issued before the interrupt is taken and any instructions that precede those instructions in the instruction stream appear to have completed before the interrupt is processed.
- No subsequent instructions in the instruction stream have begun execution.
- SRR0 addresses the instruction that would have been executed had the interrupt not occurred.

That is, these interrupts are context-synchronizing. External and decrementer interrupts are maskable. If MSR[EE] = 0, these interrupts are not recognized until EE is set. EE is cleared automatically when an interrupt is taken, to delay recognition of subsequent interrupt conditions. Interrupt handling does not begin until all currently executing instructions complete and any synchronous, precise interrupts caused by those instructions have been handled. Interrupt priorities are described in Section 6.2.5, “Interrupt Priorities.”

6.2.3 Imprecise Interrupts

The PowerPC architecture defines one imprecise interrupt, the imprecise floating-point enabled interrupt. This is implemented as one of the exception conditions that can cause a program interrupt.

6.2.3.1 Imprecise Interrupt Status Description

When the execution of an instruction causes an imprecise interrupt, SRR0 contains information related to the address of the excepting instruction as follows:

- SRR0 contains the address of either the instruction that caused the interrupt or of some instruction following that instruction.
- The interrupt is generated such that all instructions preceding the instruction addressed by SRR0 have completed with respect to the processor.
- If the imprecise interrupt is caused by the context-synchronizing mechanism (due to an instruction that caused another interrupt—for example, an alignment or data storage interrupt), SRR0 contains the address of the instruction that caused the interrupt, and that instruction may have been partially executed (refer to Section 6.2.4, “Partially Executed Instructions”).
- If the imprecise interrupt is caused by an execution-synchronizing instruction other than **sync** or **isync**, SRR0 addresses the instruction causing the interrupt. Additionally, besides causing the interrupt, that instruction is considered not to have begun execution. If the interrupt is caused by **sync** or **isync**, SRR0 may address either **sync** or **isync**, or the following instruction.
- If the imprecise interrupt is not forced by either the context- or execution-synchronizing mechanism, the instruction addressed by SRR0 is considered not to have begun execution if it did not cause the interrupt.
- When an imprecise interrupt occurs, no instruction following the instruction addressed by SRR0 is considered to have begun execution.

6.2.3.2 Recoverability of Imprecise Floating-Point Interrupts

The enabled IEEE floating-point exception mode bits, MSR[FE0,FE1], together define whether IEEE floating-point exceptions are handled precisely, imprecisely, or whether they are taken at all. The possible settings are shown in Table 6-3. For further details, see Section 3.3.6, "Floating-Point Program Exceptions."

Table 6-3. IEEE Floating-Point Program Exception Mode Bits

FE0	FE1	Mode
0	0	Floating-point exceptions ignored
0	1	Floating-point imprecise nonrecoverable. When an exception occurs, the interrupt handler is invoked at some point at or beyond the instruction that caused the exception. It may not be possible to identify the excepting instruction or the data that caused the interrupt. Results from the excepting instruction may have been used by or affected subsequent instructions executed before the interrupt handler was invoked.
1	0	Floating-point imprecise recoverable. When an exception occurs, the floating-point enabled interrupt handler is invoked at some point at or beyond the instruction that caused the interrupt. Sufficient information is provided to the interrupt handler that it can identify the excepting instruction and correct any faulty results. In this mode no incorrect results caused by the excepting instruction have been used by or affected subsequent instructions that execute before the interrupt handler is invoked.
1	1	Floating-point precise mode

Although these floating-point exceptions are maskable with FE0 and FE1, these conditions differ from other maskable interrupts in that the masking is usually controlled by the application program rather than by the operating system.

6.2.4 Partially Executed Instructions

The architecture permits certain instructions to be partially executed when an alignment or data storage interrupt occurs or an imprecise floating-point exception is forced by an instruction that causes an alignment or data storage interrupt. They are as follows:

- Load multiple/string instructions that cause an alignment or data storage interrupt—Some registers in the range of registers to be loaded may have been loaded.
- Store multiple/string instructions that cause an alignment or data storage interrupt—Some bytes in the addressed memory range may have been updated.
- Non-multiple/string store instructions that cause an alignment or data storage interrupt—Some bytes just before the boundary may have been updated. If the instruction normally alters CR0 (**stwcx.**), CR0 is set to an undefined value. For instructions that perform register updates, the update register (**r**A) is not altered.
- Floating-point load instructions that cause an alignment or data storage interrupt—The target register may be altered. For update forms, the update register (**r**A) is unchanged.

In the cases above, the number of registers and the amount of memory altered are implementation-, instruction-, and boundary-dependent. However, memory protection is not violated.

Partial execution is not allowed when integer load operations (except multiple/string operations) cause an alignment or data storage interrupt. The target register is not altered. For update forms of the integer load instructions, the update register (**r**A) is not altered.

6.2.5 Interrupt Priorities

Interrupts are roughly prioritized by class, as follows:

1. Nonmaskable, asynchronous interrupts have priority over all other interrupts—system reset and machine check interrupts (although the machine check interrupt condition can be disabled so that the condition causes the processor to go directly into the checkstop state). These two types of interrupts in this class cannot be delayed by interrupts in other classes and do not wait for the completion of any precise interrupt handling.
2. Synchronous, precise interrupts are caused by instructions and are taken in strict program order.
3. If an imprecise exception exists (the instruction that caused the interrupt has been completed and is required by the sequential execution model), interrupts signaled by instructions subsequent to the instruction that caused the interrupt are not permitted to change the architectural state of the processor. The exception causes an imprecise program interrupt unless a machine check or system reset interrupt is pending.
4. Maskable asynchronous interrupts (external interrupt and decrementer interrupts) have lowest priority.

Table 6-4 lists interrupts in order of highest-to-lowest priority.

Table 6-4. Interrupt Priorities

Class	Priority	Interrupt
Nonmaskable, asynchronous	1	System reset—The interrupt mechanism ignores all other interrupts and generates a system reset interrupt. When a system reset interrupt is generated, previously issued instructions can no longer generate interrupt conditions that cause a nonmaskable interrupt.
	2	Machine check—The interrupt mechanism ignores all other interrupts (except reset) and generates a machine check interrupt.When the machine check interrupt is generated, previously issued instructions can no longer generate exception conditions that cause a nonmaskable interrupt.

Table 6-4. Interrupt Priorities (continued)

Class	Priority	Interrupt
Synchronous, precise	3	Instruction dependent—The interrupt mechanism waits for any instructions before the excepting instruction to complete. Interrupts caused by these instructions are handled first. The appropriate interrupt is then generated if no higher priority interrupt exists. When a single instruction causes multiple interrupts, those interrupts are ordered in the following priority: A. Integer loads and stores a. Alignment b. Data storage interrupt c. Trace (if implemented) B. Floating-point loads and stores a. Floating-point unavailable b. Alignment c. Data storage interrupt d. Trace (if implemented) C. Other floating-point instructions a. Floating-point unavailable b. Program—Precise-mode floating-point enabled exception c. Floating-point assist (if implemented) d. Trace (if implemented) D. **rfi** and **mtmsr** a. Program—Privileged Instruction b. Program—Precise-mode floating-point enabled exception. These exceptions are enabled and FPSCR[FEX] is set, a program interrupt occurs no later than the next synchronizing event. c. Trace (if implemented), for **mtmsr** only E. Other instructions a. These exceptions are mutually exclusive and have the same priority: —Program: Trap —System call (**sc**) —Program: Privileged Instruction —Program: Illegal Instruction b. Trace (if implemented) F. Instruction storage interrupt. Lowest priority in this category. Recognized only when all instructions before the instruction causing this interrupt appear to have completed and that instruction is to be executed. The priority of this interrupt is specified for completeness and to ensure that it is not given more favorable treatment. An implementation can treat this interrupt as though it had a lower priority.
Imprecise	4	Program imprecise floating-point mode enabled exceptions—When this interrupt occurs, the handler is invoked at or beyond the floating-point instruction that caused the interrupt. The PowerPC architecture supports recoverable and nonrecoverable imprecise modes, which are enabled by setting MSR[FE0] ≠ MSR[FE1]. For more information see, Section 6.2.3, “Imprecise Interrupts.”
Maskable, asynchronous	5	External interrupt—The external interrupt mechanism waits for instructions currently or previously dispatched to complete execution. After all such instructions are completed, and any interrupts caused by those instructions have been handled, the interrupt mechanism generates this interrupt if no higher priority interrupt exists. This interrupt is enabled only if MSR[EE] is currently set. If EE is zero when the interrupt is detected, it is delayed until the bit is set.
	6	Decrementer—When this interrupt is created, the interrupt mechanism waits for all other possible interrupts to be reported. It then generates this interrupt if no higher priority interrupt exists. This interrupt is enabled only if MSR[EE] = 1. If EE = 0 when the interrupt is detected, it is delayed until the bit is set.

Nonmaskable, asynchronous interrupts (system reset or machine check interrupts) can occur anytime; they are not delayed if another interrupt is being handled (although machine check interrupts can be delayed by system reset interrupts). As a result, state information for the interrupted handler may be lost.

All other interrupts have lower priority than system reset and machine check interrupts, and the interrupt may not be taken immediately when it is recognized. Only one synchronous, precise interrupt can be reported at a time. If a maskable, asynchronous or an imprecise exception condition occurs while instruction-caused interrupts are being processed, its handling is delayed until all interrupts caused by previous instructions in the program flow are handled and those instructions complete execution.

6.3 Interrupt Processing

When an interrupt is taken, the processor uses SRR1 and SRR0, to save the contents of the MSR for the interrupted process and to help determine where instruction execution should resume after the interrupt is handled.

When an interrupt occurs, the address saved in SRR0 is used to help calculate where instruction processing should resume when the interrupt handler returns control to the interrupted process. Depending on the interrupt, this may be the address in SRR0 or at the next address in the program flow. All instructions in the program flow preceding this one will have completed execution and no subsequent instruction will have begun execution. This may be the address of the instruction that caused the interrupt or the next one (as in the case of a system call or trap interrupt). The SRR0 register is shown in Figure 6-1.

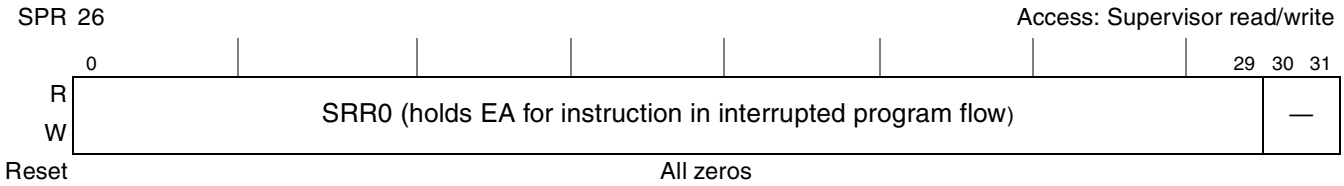


Figure 6-1. Machine Status Save/Restore Register 0 (SRR0)

The save/restore register 1 (SRR1) is used to save machine status (selected bits from the MSR and other implementation-specific status bits as well) on interrupts and to restore those values when **rfi** is executed. SRR1 is shown in Table 6-2.

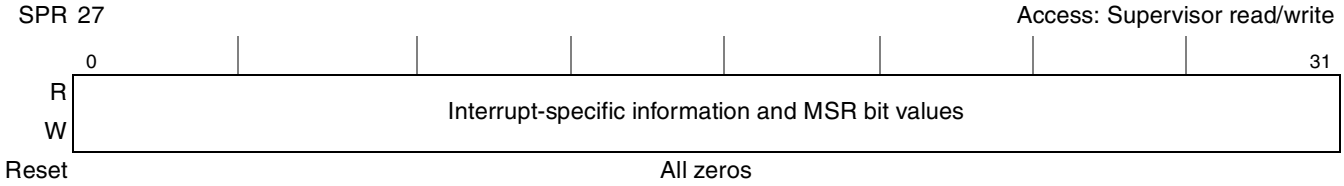


Figure 6-2. Machine Status Save/Restore Register 1 (SRR1)

When an interrupt occurs, SRR1 bits 1–4 and 10–15 are loaded with interrupt-specific information and MSR bits 16–23, 25–27, and 30–31 are placed into the corresponding bit positions of SRR1. Depending on the implementation, additional MSR bits may be copied to SRR1.

In some implementations, every instruction fetch when MSR[IR] = 1 and every data access requiring address translation when MSR[DR] = 1, may modify SRR0 and SRR1.

The MSR is 32 bits wide as shown in Figure 6-3.

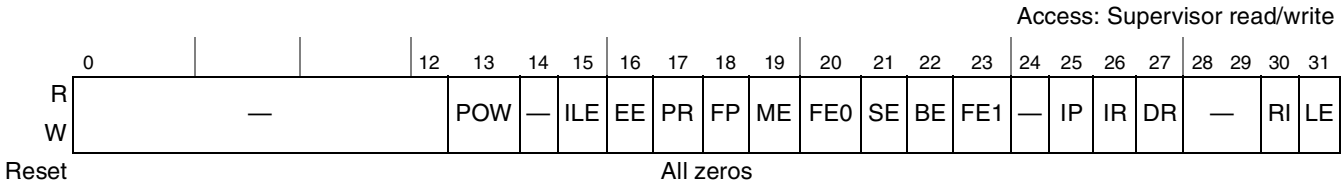


Figure 6-3. Machine State Register (MSR)

Table 6-5 shows the bit definitions for the MSR

Table 6-5. MSR Bit Settings

Bits	Name	Description
0–12	—	Reserved
13	POW	Power management enable. Power management functions are implementation-dependent. If the function is not implemented, this bit is treated as reserved. 0 Power management disabled (normal operation mode). 1 Power management enabled (reduced power mode).
14	—	Reserved, should be cleared.
15	ILE	Interrupt little-endian mode. When an interrupt occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the interrupt.
16	EE	External interrupt enable 0 The processor delays recognition of external and decrementer interrupt conditions. 1 The processor is enabled to take an external interrupt or the decrementer interrupt.
17	PR	Privilege level 0 The processor can execute both user- and supervisor-level instructions. 1 The processor can only execute user-level instructions.
18	FP	Floating-point available 0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves. 1 The processor can execute floating-point instructions.
19	ME	Machine check enable 0 Machine check interrupts are disabled. 1 Machine check interrupts are enabled.
20	FE0	Floating-point interrupt mode 0 (see Table 2-10).
21	SE	Single-step trace enable (Optional) 0 The processor executes instructions normally. 1 The processor generates a single-step trace interrupt upon the successful execution of the next instruction. Note: If the function is not implemented, this bit is treated as reserved.
22	BE	Branch trace enable (Optional) 0 The processor executes branch instructions normally. 1 The processor generates a branch trace interrupt after completing the execution of a branch instruction, regardless of whether or not the branch was taken. Note: If the function is not implemented, this bit is treated as reserved.
23	FE1	Floating-point exception mode 1 (See Table 2-10).

Table 6-5. MSR Bit Settings (continued)

Bits	Name	Description
24	—	Reserved, should be cleared.
25	IP	Interrupt prefix. The setting of this bit specifies whether an interrupt vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the interrupt vector. See Table 6-2. 0 Interrupts are vectored to the physical address 0x000*n_nnnn* . 1 Interrupts are vectored to the physical address 0xFFF*n_nnnn*. In most systems, IP is set during system initialization and cleared when initialization is complete.
26	IR	Instruction address translation 0 Instruction address translation is disabled. 1 Instruction address translation is enabled. For more information see Chapter 7, “Memory Management.”
27	DR	Data address translation 0 Data address translation is disabled. 1 Data address translation is enabled. For more information see Chapter 7, “Memory Management.”
28–29	—	Reserved, should be cleared.
30	RI	Recoverable interrupt (for system reset and machine check interrupts). 0 Interrupt is not recoverable. 1 Interrupt is recoverable. For more information see Section 6.5.1, “System Reset Interrupt (0x00100),” and Section 6.5.2, “Machine Check Interrupt (0x00200).”
31	LE	Little-endian mode enable 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

Those MSR bits that are written to SRR1 are written when the first instruction of the interrupt handler is encountered. The data address register (DAR) is used by several interrupts (for example, alignment and data storage interrupts) to identify the address of a memory element.

6.3.1 Enabling and Disabling Interrupts

When a condition exists that may cause an interrupt to be generated, it must be determined whether the interrupt is enabled for that condition as follows:

- IEEE floating-point enabled exceptions (which generate program interrupt) are ignored when both MSR[FE0] and MSR[FE1] are cleared. If either of these bits is set, all IEEE enabled floating-point exceptions are taken and cause a program interrupt.
- Asynchronous, maskable interrupts (that is, the external and decrementer interrupts) are enabled by setting MSR[EE]. When MSR[EE] = 0, recognition of these interrupt conditions is delayed. MSR[EE] is cleared automatically when an interrupt is taken to delay recognition of exceptions causing those interrupts.
- A machine check interrupt can only occur if the machine check enable bit, MSR[ME], is set. If MSR[ME] is cleared, the processor goes directly into checkstop state when a machine check exception condition occurs.

6.3.2 Steps for Interrupt Processing

After it is determined that the interrupt can be taken (by confirming that any instruction-caused interrupts occurring earlier in the instruction stream have been handled, and by confirming that the interrupt is enabled for the interrupt condition), the processor does the following:

1. The machine status save/restore register 0 (SRR0) is loaded with an instruction address that depends on the type of interrupt. See the individual interrupt description for details about how this register is used for specific interrupts.
2. SRR1 bits 1–4 and 10–15 are loaded with interrupt-specific information.
3. MSR bits 16–23, 25–27, and 30–31 are loaded with a copy of the corresponding MSR bits. Note that some implementations save additional MSR bits to SRR1.
4. The MSR is set as described in Table 6-6. The new values take effect beginning with the fetching of the first instruction of the interrupt handler located at the interrupt vector address.

   Note that MSR[IR,DR] are cleared for all interrupt types; therefore, address translation is disabled for both instruction fetches and data accesses beginning with the first instruction of the interrupt handler.

   Also, note that the MSR[ILE] setting at the time of the interrupt is copied to MSR[LE] when the interrupt is taken (as shown in Table 6-6).
5. Instruction fetch and execution resumes, using the new MSR value, at a location specific to the interrupt type. The location is determined by adding the interrupt's vector offset (see Table 6-2) to the base address determined by MSR[IP]. If IP is cleared, interrupts are vectored to the physical address 0x000*n_nnnn*. If IP is set, interrupts are vectored to the physical address 0xFFF*n_nnnn*. For a machine check interrupt that occurs when MSR[ME] = 0 (machine check interrupts are disabled), the checkstop state is entered (the machine stops executing instructions). See Section 6.5.2, “Machine Check Interrupt (0x00200).”

In some implementations, any instruction fetch with MSR[IR] = 1 and any load or store with MSR[DR] = 1 may cause SRR0 and SRR1 to be modified.

6.3.3 Returning from an Interrupt Handler

The Return from Interrupt (**rfi**) instruction performs context synchronization by allowing previously issued instructions to complete before returning to the interrupted process. Execution of the **rfi** instruction ensures the following:

- All previous instructions have completed to a point where they can no longer cause an interrupt.
- Previous instructions complete execution in the context (privilege, protection, and address translation) under which they were issued.
- The **rfi** instruction copies SRR1 bits back into the MSR.
- Subsequent instructions execute in the context established by this instruction.

See Section 6.2.2, “Context Synchronization.”

6.4 Process Switching

The operating system should execute the following when processes are switched:

- A **sync** instruction, which orders the effects of instruction execution. All instructions previously initiated appear to have completed before the **sync** instruction completes, and no subsequent instructions appear to be initiated until the **sync** instruction completes.
- An **isync** instruction, which waits for all previous instructions to complete and then discards any fetched instructions, causing subsequent instructions to be fetched (or refetched) from memory and to execute in the context (privilege, translation, protection, etc.) established by the previous instructions.
- A **stwcx.** instruction, which clears any outstanding reservations and ensures that a **lwarx** instruction in the old process is not paired with an **stwcx.** instruction in the new process.

The operating system should handle MSR[RI] as follows:

- In machine check and system reset interrupt handlers—If the SRR1 bit corresponding to MSR[RI] is cleared, the interrupt is not recoverable.
- In each interrupt handler—When enough state information is saved that a machine check or system reset interrupt can reconstruct the previous state, set MSR[RI].
- At the end of each interrupt handler—Clear MSR[RI], set SRR0 and SRR1 appropriately, and then execute **rfi**.

Note that the RI bit being set indicates that, with respect to the processor, enough processor state data is valid for the processor to continue, but it does not guarantee that the interrupted process can resume.

6.5 Interrupt Definitions

Table 6-7 shows all interrupt types and certain MSR bit settings when the interrupt handler is invoked. Depending on the interrupt, some of these bits are stored in SRR1 when an interrupt is taken. The following subsections describe each interrupt in detail.

Table 6-7. MSR Setting Due to Interrupt

Interrupt Type	MSR Bit														
	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IP	IR	DR	RI	LE
System reset	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Machine check	0	—	0	0	0	0	0	0	0	0	—	0	0	0	ILE
Data access	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Instruction access	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
External	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Alignment	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Program	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Floating-point unavailable	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Decrementer	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE

Table 6-7. MSR Setting Due to Interrupt (continued)

Interrupt Type	MSR Bit														
	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IP	IR	DR	RI	LE
System call	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Trace	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE
Floating-point assist	0	—	0	0	0	—	0	0	0	0	—	0	0	0	ILE

0 Bit is cleared
1 Bit is set
ILE Bit is copied from MSR[ILE]
— Bit is not altered
Reading of reserved bits may return 0, even if the value last written to it was 1.

6.5.1 System Reset Interrupt (0x00100)

The system reset interrupt is a nonmaskable, asynchronous interrupt signaled to the processor typically through the assertion of a system-defined signal; see Table 6-8.

Table 6-8. System Reset Interrupt—Register Settings

Register	Setting Description														
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.														
SRR1	1–4,10–15 Cleared 16–23, 25–27, 31 Loaded with equivalent bits from the MSR 30 Loaded from the equivalent MSR bit, MSR[RI], if the interrupt is recoverable; otherwise cleared. **Note:** Depending on the implementation, additional MSR bits may be copied to SRR1. If the processor state is corrupted to the extent that execution cannot resume reliably, the bit corresponding to MSR[RI], (SRR1[30]), is cleared.														
MSR	POW	ILE	EE	PR	FP	ME	FE0	SE	BE	FE1	IP	IR	DR	RI	LE
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

When a system reset interrupt is taken, instruction execution continues at offset 0x00100 from the physical base address determined by MSR[IP].

If the interrupt is recoverable, the value of MSR[RI] is copied to the corresponding SRR1 bit. The interrupt functions as a context-synchronizing operation. The interrupt is not recoverable if a reset interrupt causes the loss of the following:

- An external interrupt (interrupt or decrementer)
- Floating-point enabled type program interrupt

If the SRR1 bit corresponding to MSR[RI] is cleared, the interrupt is context-synchronizing only with respect to subsequent instructions. Note that each implementation provides a means for software to distinguish between power-on reset and other types of system resets (such as soft reset).

6.5.2 Machine Check Interrupt (0x00200)

If no higher-priority interrupt is pending (namely, a system reset interrupt), the processor initiates a machine check interrupt when the appropriate condition is detected. Note that the causes of machine check interrupts are implementation- and system-dependent, and are typically signalled to the processor by the assertion of a specified signal on the processor interface.

When a machine check condition occurs and MSR[ME] = 1, the interrupt is recognized and handled. If MSR[ME] = 0 and a machine check occurs, the processor generates an internal checkstop condition. When a processor is in checkstop state, instruction processing is suspended and generally cannot continue without resetting the processor. Some implementations may preserve some or all of the internal state when entering checkstop state, so that the state can be analyzed as an aid in problem determination.

In general, it is expected that a bus error signal would be used by a memory controller to indicate a memory parity error or an uncorrectable memory ECC error. Note that the resulting machine check interrupt has priority over any interrupts caused by the instruction that generated the bus operation.

If a machine check exception causes an interrupt that is not context-synchronizing, the interrupt is not recoverable. Also, a machine check interrupt is not recoverable if it causes the loss of one of the following:

- An external interrupt (interrupt or decrementer)
- Floating-point enabled type program interrupt

If the SRR1 bit corresponding to MSR[RI] is cleared, the interrupt is context-synchronizing only with respect to subsequent instructions. If the interrupt is recoverable, the SRR1 bit corresponding to MSR[RI] is set and the interrupt is context-synchronizing.

Note that if the error is caused by the memory subsystem, incorrect data could be loaded into the processor and register contents could be corrupted regardless of whether the interrupt is considered recoverable by the SRR1 bit corresponding to MSR[RI].

On some implementations (see individual reference manuals for device-specific information), a machine check interrupt may be caused by referring to a nonexistent physical address, either because translation is disabled (MSR[IR] or MSR[DR] = 0) or through an invalid translation. On such a system, execution of **dcbz** or **dcba** can cause a delayed machine check interrupt by introducing a block into the data cache that is associated with an invalid physical address. A machine check interrupt could eventually occur when and if a subsequent attempt is made to store that block to memory (for example, as the block becomes the target for replacement, or as the result of executing a **dcbst** instruction).

When a machine check interrupt is taken, registers are updated as shown in Table 6-9.

Table 6-9. Machine Check Interrupt—Register Settings

Register	Setting Description														
SRR0	On a best-effort basis, implementations can set this to an EA of some instruction that was executing or about to be executing when the machine check condition occurred.														
SRR1	Bit 30 is loaded from MSR[RI] if the processor is in a recoverable state. Otherwise cleared. The setting of all other SRR1 bits is implementation-dependent.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME** [1]	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	0	0	0	0	0	—	0	0	0	Set to value of ILE

[1] Note that when a machine check interrupt is taken, the interrupt handler should set MSR[ME] as soon as it is practical to handle another machine check interrupt. Otherwise, subsequent machine check interrupts cause the processor to automatically enter the checkstop state.

If MSR[RI] is set, the machine check interrupt may still be unrecoverable in the sense that execution cannot resume in the same context that existed before the interrupt.

6.5.3 Data Storage Interrupt (0x00300)

A data storage interrupt (DSI) occurs when no higher priority interrupt exists and a data memory access cannot be performed. The condition that caused the DSI can be determined by reading the DSISR, a supervisor-level SPR (SPR18) that can be read by using the **mfspr** instruction. Bit settings are provided in Table 6-10. Table 6-10 also indicates which memory element is pointed to by the DAR. DSIs can be generated by load/store instructions, cache-control instructions (**icbi**, **dcbi**, **dcbz**, **dcbst**, and **dcbf**), or the **eciwx**/**ecowx** instructions for any of the following reasons:

- The effective address cannot be translated. That is, there is a page fault for this portion of the translation, so a DSI must be taken to retrieve the translation, for example from a storage device such as a hard disk drive.
- The instruction is not supported for the type of memory addressed.
  - For **lwarx**/**stwcx.** instructions that reference a memory location that is write-through-required. If the interrupt is not taken, the instructions execute correctly.
- The access violates memory protection.
- The execution of an **eciwx** or **ecowx** instruction is disallowed because the external access register enable bit (EAR[E]) is cleared.
- A data address breakpoint register (DABR) match occurs. The DABR facility is optional to the PowerPC architecture, but if one is implemented, it is recommended that it be implemented as follows. A data address breakpoint match is detected for a load or store instruction if the three following conditions are met for any byte accessed:
  - EA[0–28] = DABR[DAB]
  - MSR[DR] = DABR[BT]
  - The instruction is a store and DABR[DW] = 1, or it is a load and DABR[DR] = 1.

  The DABR is described in Section 2.3.14, “Data Address Breakpoint Register (DABR).”

DAR settings are described in Table 6-10. If the above conditions are satisfied, it is undefined whether a match occurs in the following cases:

— The instruction is store conditional but the store is not performed.

— The instruction is a load/store string of zero length.

— The instruction is **dcbz**, **eciwx**, or **ecowx**

The cache management instructions other than **dcbz** never cause a match. If **dcbz** causes a match, some or all of the target memory locations may have been updated. For the purpose of determining whether a match occurs, **eciwx** is treated as a load, and **ecowx** and **dcbz** are treated as stores.

If an **stwcx.** instruction has an EA for which a normal store operation would cause a data storage interrupt but the processor does not have the reservation from **lwarx**, whether a data storage interrupt is taken is implementation-dependent.

If the value in XER[25–31] indicates that a load or store string instruction has a length of zero, a data storage interrupt does not occur, regardless of the effective address.

The interrupt condition that caused the interrupt is defined in the DSISR.

Table 6-10. Data Storage Interrupt (DSI)—Register Settings

Register	Setting Description														
SRR0	Set to the effective address of the instruction that caused the interrupt.														
SRR1	1–4,10–15 Cleared 16–23, 25–27,30–31 Loaded with equivalent bits from the MSR **Note:** Depending on the implementation, additional MSR bits may be copied to SRR1.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

Table 6-10. Data Storage Interrupt (DSI)—Register Settings (continued)

Register	Setting Description
DSISR	0 Set if a load or store instruction results in a direct-store error interrupt; otherwise cleared. 1 Set if the translation of an attempted access is not found in the primary or secondary hash table entry group (HTEG), or in the range of a DBAT register (page fault condition); otherwise cleared. 2–3 Cleared 4 Set if a memory access is not permitted by the page or DBAT protection mechanism; otherwise cleared. 5 Set if an **eciwx**, **ecowx**, **lwarx,** or **stwcx.** is attempted to direct-store interface space, or **lwarx** or **stwcx.** is used with addresses marked as write-through. Otherwise cleared. 6 Set for a store operation and cleared for a load. 7–8 Cleared 9 Set if a DABR match occurs. Otherwise cleared. 10 Cleared 11 Set if the instruction is an **eciwx** or **ecowx** and EAR[E] = 0; otherwise cleared. 12–31 Cleared Due to the multiple interrupt conditions possible from the execution of a single instruction, the following combinations of DSISR bits may be set concurrently: Bits 1 and 11, Bits 4 and 5, Bits 4 and 11, Bits 5 and 11. Additonally, bit 6 is set if the instruction that caused the interrupt is a store, **ecowx**, **dcbz**, **dcba**, or **dcbi** and bit 6 would otherwise be cleared. Also, bit 9 (DABR match) may be set alone, in combination with any other bit, or with any of the other combinations shown above.
DAR	Set to the effective address of a memory element as described in the following list: • A byte in the first word accessed in the segment or BAT area that caused the DSI, for a byte, half word, or word memory access (to a segment or BAT area). • A byte in the first double word accessed in the segment or BAT area that caused the DSI, for a double-word memory access (to a segment or BAT area). • A byte in the block that caused the interrupt for a cache management instruction. • The EA computed by the instruction for the attempted execution of **eciwx** or **ecowx** when EAR[E] is cleared. If the interrupt is caused by a DABR match, the DAR is set to the EA of any byte in the range from A to B inclusive, where A is the EA of the word (for a byte, half word, or word access) or double word (for a double-word access) specified by the EA computed by the instruction, and B is the EA of the last byte in the word or double word in which the match occurred.

6.5.4 Instruction Storage Interrupt (0x00400)

An instruction storage interrupt (ISI) occurs when no higher priority interrupt exists and an attempt to fetch the next instruction to be executed fails for any of the following reasons:

- The effective address cannot be translated. For example, when there is a page fault for this portion of the translation, an ISI must be taken to retrieve the page (and possibly the translation), typically from a storage device.
- An attempt is made to fetch an instruction from a no-execute segment.
- An attempt is made to fetch an instruction from guarded memory and MSR[IR] = 1.
- The fetch access violates memory protection.

Register settings for ISI are shown in Table 6-11.

Table 6-11. Instruction Storage Interrupt (ISI)—Register Settings

Register	Setting Description														
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present (if the interrupt occurs on attempting to fetch a branch target, SRR0 holds the target address).														
SRR1	1 2 3 4 10–15 16–23, 25–27, 30–31 Note that only one of bits 1, 3, and 4 can be set. Also, note that depending on the implementation, additional MSR bits may be copied to SRR1.	Set if the translation of an attempted access is not found in the primary or secondary hash table entry group (HTEG) or in the range of an IBAT register (page fault condition); otherwise cleared. Cleared Set if the fetch access occurs to a direct-store segment (SR[T] = 1), to a no-execute segment (N bit set in segment descriptor), or to guarded memory when MSR[IR] = 1. Otherwise, cleared. Note that the direct-store facility is being phased out of the architecture and is not likely to be supported in future devices. Set if a memory access is not permitted by the page or IBAT protection mechanism, described in Section 7.5.4, “Block Memory Protection,” and Section 7.6.4, “Page Memory Protection”; otherwise cleared. Cleared Loaded with equivalent bits from the MSR													
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

6.5.5 External Interrupt (0x00500)

An external interrupt is signaled to the processor by the assertion of the external interrupt signal. The interrupt may be delayed by other higher priority interrupts or if MSR[EE] is zero when the interrupt is detected. Note that the occurrence of this interrupt does not cancel the external request.

The register settings for the external interrupt are shown in Table 6-12.

Table 6-12. External Interrupt—Register Settings

Register	Setting Description													
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.													
SRR1	1–410–15 16–23,25–2730–31 Note that depending on the implementation, additional MSR bits may be copied to SRR1.	Cleared Loaded with equivalent bits from the MSR												
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

When an external interrupt is taken, instruction execution resumes at offset 0x00500 from the physical base address determined by MSR[IP].

6.5.6 Alignment Interrupt (0x00600)

This section describes conditions that can cause alignment interrupts. Similar to data storage interrupts, alignment interrupts use the SRR0 and SRR1 to save the machine state and the DSISR to determine the source of the interrupt. An alignment interrupt occurs when no higher priority interrupt exists and the implementation cannot perform a memory access for one of the following reasons:

- An operand of a floating-point load or store instruction is not word-aligned.
- The operand of **lmw**, **stmw**, **lwarx**, **stwcx.**, **eciwx**, or **ecowx** is not aligned.
- The instruction is **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** and the processor is in little-endian mode.
- An operand of an elementary or string load or store crosses a protection boundary.
- An operand of **lmw** or **stmw** crosses a segment or BAT boundary.
- An operand of **dcbz** is in memory that is write-through-required or caching inhibited, or **dcbz** is executed in an implementation that has either no data cache or a write-through data cache.

For **lmw**, **stmw**, **lswi**, **lswx**, **stswi**, and **stswx** instructions in little-endian mode, an alignment interrupt always occurs. For **lmw** and **stmw** instructions with an operand that is not aligned in big-endian mode, and for **lwarx**, **stwcx.**, **eciwx**, and **ecowx** with an operand that is not aligned in either endian mode, an implementation may yield boundedly-undefined results instead of causing an alignment interrupt (for **eciwx** and **ecowx** when EAR[E] = 0, a third alternative is to cause a data storage interrupt). For all other cases listed above, an implementation may execute the instruction correctly instead of causing an alignment interrupt. For the **dcbz** instruction, correct execution means clearing each byte of the block in main memory. See Section 3.1, “Data Organization in Memory and Data Transfers,” for a complete definition of alignment in the PowerPC architecture.

The term, ‘protection boundary’, refers to the boundary between protection domains. A protection domain is a segment, a block of memory defined by a BAT entry, a virtual 4-Kbyte page, or a range of unmapped effective addresses. Protection domains are defined only when the corresponding address translation (instruction or data) is enabled (MSR[IR] or MSR[DR] = 1).

The register settings for alignment interrupts are shown in Table 6-13.

Table 6-13. Alignment Interrupt—Register Settings

Register	Setting Description														
SRR0	Set to the effective address of the instruction that caused the interrupt.														
SRR1	1–4,10–15 Cleared 16–23, 25–27,30–31 Loaded with equivalent bits from the MSR Note that depending on the implementation, addition MSR bits may be copied to SRR1														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

Table 6-13. Alignment Interrupt—Register Settings (continued)

Register	Setting Description
DSISR	0–14 Cleared 15–16 For register indirect with index addressing—set to bits 29–30 of the instruction encoding. For register indirect with immediate index addressing—cleared 17 For register indirect with index addressing—set to bit 25 of the instruction encoding. For register indirect with immediate index addressing— set to bit 5 of the instruction encoding. 18–21 For register indirect with index addressing—set to bits 21–24 of the instruction encoding. For register indirect with immediate index addressing—set to bits 1–4 of the instruction encoding. 22–26 Set to bits 6–10 (identifying either the source or destination) of the instruction encoding. Undefined for **dcbz**. 27–31 Set to bits 11–15 of the instruction encoding (**r**A) for update-form instructions Set to either bits 11–15 of the instruction encoding or to any register number not in the range of registers loaded by a valid form instruction for **lmw**, **lswi**, and **lswx** instructions. Otherwise undefined. **Note:** For load or store instructions that use register indirect with index addressing, the DSISR can be set to the same value that would have resulted if the corresponding instruction uses register indirect with immediate index addressing had caused the interrupt. Similarly, for load or store instructions that use register indirect with immediate index addressing, DSISR can hold a value that would have resulted from an instruction that uses register indirect with index addressing. For example, a misaligned **lwarx** instruction that crosses a protection boundary would normally cause the DSISR to be set to the following binary value:000000000000 00 0 01 0 0101 ttttt ????? The value ttttt refers to the destination and ????? indicates undefined bits. However, this register may be set as if the instruction were **lwa**, as follows: 000000000000 10 0 00 0 1101 ttttt ????? If there is no corresponding instruction, no alternative value can be specified. The following instruction pairs can use the same DSISR values: **lbz/lbzx, lbzu/lbzux, lhz/lhzx, lhzu/lhzux, lha/lhax, lhau/lhaux, lwz/lwzx, lwzu/lwzux, lwa/lwax, stb/stbx, stbu/stbux, sth/sthx, sthu/sthux, stw/stwx, stwu/stwux, lfs/lfsx, lfsu/lfsux, stfs/stfsx, stfsu/stfsux**
DAR	Set to the EA of the data access as computed by the instruction causing the alignment interrupt.

The architecture does not support the use of a misaligned EA by load/store with reservation instructions or by the **eciwx** and **ecowx** instructions. If one of these instructions specifies a misaligned EA, the interrupt handler should not emulate the instruction but should treat the occurrence as a programming error.

6.5.6.1 Integer Alignment Interrupts

Operations that are not naturally aligned may suffer performance degradation, depending on the processor design, the type of operation, the boundaries crossed, and the mode that the processor is in during execution. More specifically, these operations may either cause an alignment interrupt or they may cause the processor to break the memory access into multiple, smaller accesses with respect to the cache and the memory subsystem.

6.5.6.1.1 Page Address Translation Access Considerations

A page address translation access occurs when MSR[DR] is set, SR[T] is cleared, and there is no BAT match. Note that **dcbz** causes an alignment interrupt if the access is to a location with the W (write-through) or I (cache-inhibit) bit set.

Misaligned memory accesses that do not cause an alignment interrupt may not perform as well as an aligned access of the same type. The resulting performance degradation due to misaligned accesses depends on how well each individual access behaves with respect to the memory hierarchy.

Details regarding page address translation is implementation-dependent and is found in an implementation's reference manual.

6.5.6.2 Little-Endian Mode Alignment Interrupts

The OEA allows implementations to take alignment interrupts on misaligned accesses (as described in Section 3.1.4, "Byte Ordering in PowerPC Architecture") in little-endian mode, but does not require them to do so. Some implementations may perform some misaligned accesses without taking an interrupt.

6.5.6.3 Interpretation of the DSISR as Set by an Alignment Interrupt

For most alignment interrupts, an interrupt handler can emulate the instruction that causes the interrupt. To do this, the handler requires the following characteristics of the instruction:

- Load or store
- Length (half word or word)
- String, multiple, or normal load/store
- Integer or floating-point
- Whether the instruction performs update
- Whether the instruction performs byte reversal
- Whether it is a **dcbz** instruction

The PowerPC architecture provides this information implicitly by setting DSISR bits that identify the excepting instruction type. The handler does not need to load the excepting instruction from memory. The mapping for all interrupt possibilities is unique except for the few interrupts discussed below.

Table 6-14 shows the inverse mapping—how the DSISR bits identify the instruction that caused the interrupt.

The alignment interrupt handler cannot distinguish a floating-point load or store that causes an interrupt because it is misaligned. However, this does not matter; in either case, it is emulated with integer instructions.

Table 6-14. DSISR [15–21] Settings to Determine Misaligned Instruction

DSISR[15–21]	Instruction	DSISR[15–21]	Instruction
00 0 0000	**lwarx**, **lwz**, special cases[1]	01 1 0010	—
00 0 0010	—	01 1 0101	**lwaux**
00 0 0010	**stw**	10 0 0010	**stwcx.**
00 0 0100	**lhz**	10 0 0011	—
00 0 0101	**lha**	10 0 1000	**lwbrx**
00 0 0110	**sth**	10 0 1010	**stwbrx**
00 0 0111	**lmw**	10 0 1100	**lhbrx**
00 0 1000	**lfs**	10 0 1110	**sthbrx**
00 0 1001	—	10 1 0100	**eciwx**

Table 6-14. DSISR [15–21] Settings to Determine Misaligned Instruction (continued)

DSISR[15–21]	Instruction	DSISR[15–21]	Instruction
00 0 1010	**stfs**	10 1 0110	**ecowx**
00 0 1011	—	10 1 1111	**dcbz**
00 0 1101	**ld**,**lwa** [2]	11 0 0000	**lwzx**
00 0 1111	**std**	11 0 0010	**stwx**
00 1 0000	**lwzu**	11 0 0100	**lhzx**
00 1 0010	**stwu**	11 0 0101	**lhax**
00 1 0100	**lhzu**	11 0 0110	**sthx**
00 1 0101	**lhau**	11 0 1000	**lfsx**
00 1 0110	**sthu**	11 0 1001	—
00 1 0111	**stmw**	11 0 1010	**stfsx**
00 1 1000	**lfsu**	11 0 1011	—
00 1 1001	—	11 0 1111	**stfiwx**
00 1 1010	**stfsu**	11 1 0000	**lwzux**
00 1 1011	—	11 1 0010	**stwux**
01 0 0000	—	11 1 0100	**lhzux**
01 0 0010	—	11 1 0101	**lhaux**
01 0 0101	**lwax**	11 1 0110	**sthux**
01 0 1000	**lswx**	11 1 1000	**lfsux**
01 0 1001	**lswi**	11 1 1001	—
01 0 1010	**stswx**	11 1 1010	**stfsux**
01 0 1011	**stswi**	11 1 1011	—
01 1 0000	—	—	—

[1] The instructions **lwz** and **lwarx** give the same DSISR bits (all zero). But if **lwarx** causes an alignment interrupt, it is an invalid form, so it need not be emulated in any precise way. It is adequate for the alignment interrupt handler to simply emulate the instruction as if it were an **lwz**. It is important that the emulator use the address in the DAR, rather than computing it from **r**A/**r**B/D, because **lwz** and **lwarx** use different addressing modes.

If opcode 0 (illegal or reserved) can cause an alignment interrupt, the interrupt handler cannot distinguish it from **lwarx** and **lwz**.

[2] These instructions are distinguished by DSISR[12–13], which are not shown in this table.

6.5.7 Program Interrupt (0x00700)

A program interrupt occurs when no higher priority interrupt exists and one or more of the following interrupts conditions, which correspond to bit settings in SRR1, occur during execution of an instruction:

- System IEEE floating-point enabled exception—A system IEEE floating-point enabled exception can be generated when FPSCR[FEX] is set and either (or both) MSR[FE0] or MSR[FE1] is set.

FPSCR[FEX] is set by the execution of a floating-point instruction that causes an enabled exception or by the execution of a ‘move to FPSCR’ type instruction that sets an exception bit when its corresponding enable bit is set. Floating-point interrupts are described in Section 3.3.6, “Floating-Point Program Exceptions.”

- Illegal instruction—An illegal instruction program interrupt is generated when execution of an instruction is attempted with an illegal opcode or illegal combination of opcode and extended opcode fields (these include unimplemented PowerPC instructions), or when execution of an unimplemented optional or a reserved instruction is attempted.

  Note that implementations are permitted to generate an illegal instruction program interrupt when encountering the following instructions. If an illegal instruction interrupt is not generated, the alternative is shown in parenthesis.

  — An instruction corresponds to an invalid class (the results may be boundedly undefined)
  — An **lswx** instruction for which **r**A or **r**B is in the range of registers to be loaded (may cause results that are boundedly undefined)
  — A move to/from SPR instruction with an SPR field that does not contain one of the defined values, as follows:
    – MSR[PR] = 1 and SPR[0] = 1 (can cause a privileged instruction program interrupt)
    – MSR[PR] = 0 or SPR[0] = 0 (may cause boundedly-undefined results)
  — An unimplemented floating-point instruction that is not optional (may cause a floating-point assist interrupt)
- Privileged instruction—A privileged instruction type program interrupt is generated when the execution of a privileged instruction is attempted and the processor is operating in user mode (MSR[PR] is set). It is also generated for **mtspr** or **mfspr** instructions that have an invalid SPR field that contain one of the defined values having SPR[0] = 1 and if MSR[PR] = 1. Some implementations may also generate a privileged instruction program interrupt if a specified SPR field (for a move to/from SPR instruction) is not defined for a particular implementation, but SPR[0] = 1; in this case, the implementation may cause either a privileged instruction program interrupt, or an illegal instruction program interrupt may occur instead.
- Trap—A trap program interrupt is generated when any of the conditions specified in a trap instruction is met. Trap instructions are described in Section 4.2.4.6, “Trap Instructions.”

The register settings when a program interrupt is taken are shown in Table 6-15.

Table 6-15. Program Interrupt—Register Settings

Register	Setting Description														
SRR0	The contents of SRR0 differ according to the following situations: • For all program interrupts except those caused by floating-point enabled exceptions when operating in imprecise mode (MSR[FE0] ≠ MSR[FE1]), SRR0 contains the EA of the excepting instruction. • When the processor is in floating-point imprecise mode, SRR0 may contain the EA of the excepting instruction or that of a subsequent unexecuted instruction. If the subsequent instruction is **sync** or **isync**, SRR0 points no more than four bytes beyond the **sync** or **isync** instruction. • If FPSCR[FEX] = 1, but IEEE floating-point enabled exceptions are disabled (MSR[FE0] = MSR[FE1] = 0), the program interrupt occurs before the next synchronizing event if an instruction alters those bits (thus enabling the program interrupt). When this occurs, SRR0 points to the instruction that would have executed next and not to the instruction that modified MSR.														
SRR1	1–4, 10 Cleared 11 Set for an IEEE floating-point enabled exception; otherwise cleared. 12 Set for an illegal instruction exception; otherwise cleared. 13 Set for a privileged instruction exception ; otherwise cleared. 14 Set for a trap interrupt; otherwise cleared. 15 Cleared if SRR0 contains the address of the instruction causing the interrupt, and set if SRR0 contains the address of a subsequent instruction. 16–23, 25–27 Loaded with equivalent bits from the MSR 30–31 Note that depending on the implementation, additional MSR bits may be copied to SRR1.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

When a program interrupt is taken, instruction execution resumes at offset 0x00700 from the physical base address determined by MSR[IP].

6.5.8 Floating-Point Unavailable Interrupt (0x00800)

A floating-point unavailable interrupt occurs when no higher priority interrupt exists, an attempt is made to execute a floating-point instruction (including floating-point load, store, or move instructions), and the floating-point available bit in the MSR is cleared, (MSR[FP] = 0).

The register settings for floating-point unavailable interrupts are shown in Table 6-16.

Table 6-16. Floating-Point Unavailable Interrupt—Register Settings

Register	Setting Description														
SRR0	Set to the effective address of the instruction that caused the interrupt.														
SRR1	1–4, 10–15 Cleared 16–23, 25–27 Loaded with equivalent bits from the MSR 30–31 Note that depending on the implementation, additional MSR bits may be copied to SRR1.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

6.5.9 Decrementer Interrupt (0x00900)

A decrementer interrupt occurs when no higher priority interrupt exists, a decrementer interrupt condition occurs (for example, the decrementer register has completed decrementing), and MSR[EE] = 1. The decrementer register counts down, causing an interrupt request when it passes through zero. A decrementer interrupt request remains pending until the decrementer interrupt is taken and then it is cancelled. The decrementer implementation meets the following requirements:

- The decrementer and the time-base counter are driven by the same fundamental time base.
- Loading a GPR from the DEC does not affect the decrementer.
- Storing a GPR value to the DEC replaces the DEC value with the value in the GPR.
- Whenever DEC[0] changes from 0 to 1, a decrementer interrupt request is signaled. If multiple decrementer interrupt requests are received before the first can be reported, only one interrupt is reported. The occurrence of a decrementer interrupt cancels the request.
- If DEC is altered by software and if bit 0 is changed from 0 to 1, an interrupt request is signaled.

The register settings for the decrementer interrupt are shown in Table 6-17.

Table 6-17. Decrementer Interrupt—Register Settings

Register	Setting Description														
SRR0	Set to the effective address of the instruction that the processor would have attempted to execute next if no interrupt conditions were present.														
SRR1	1–4, 10–15 Cleared 16–23, 25–27 Loaded with equivalent bits from the MSR 30–31 Note that depending on the implementation, additional MSR bits may be copied to SRR1.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

6.5.10 System Call Interrupt (0x00C00)

A system call interrupt occurs when a System Call (**sc**) instruction is executed. The effective address of the instruction following the **sc** instruction is placed into SRR0. MSR bits are saved in SRR1, as shown in Table 6-18. Then a system call interrupt is generated.

The system call interrupt causes the next instruction to be fetched from offset 0x00C00 from the physical base address determined by the new setting of MSR[IP]. As with most other interrupts, this interrupt is context-synchronizing. Section 6.2.2, “Context Synchronization,” gives more information on the actions performed by a context-synchronizing operation. Register settings are shown in Table 6-18.

Table 6-18. System Call Interrupt—Register Settings

Register	Setting Description														
SRR0	Set to the effective address of the instruction following the System Call instruction														
SRR1	1–4, 10–15 Cleared 16–23, 25–27, 30–31 Loaded with equivalent bits from the MSR Note that depending on the implementation, additional MSR bits may be copied to SRR1.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

6.5.11 Trace Interrupt (0x00D00)

The trace interrupt is optional to the PowerPC architecture, and specific information about how it is implemented can be found in user's manuals for individual processors.

The trace interrupt provides a means of tracing the flow of control of a program for debugging and performance analysis purposes. It is controlled by MSR[SE,BE] as follows:

- MSR[SE] = 1: the processor generates a single-step type trace interrupt after each instruction that completes without causing an interrupt or context change (such as occurs when an **sc**, **rfi**, or a load instruction that causes an interrupt, for example, is executed).
- MSR[BE] = 1: the processor generates a branch-type trace interrupt after completing the execution of a branch instruction, regardless of whether the branch is taken.

If this facility is implemented, a trace interrupt occurs when no higher priority interrupt exists and either of the conditions described above exist. The following are not traced:

- **rfi** instruction
- **sc** and trap instructions that trap
- Other instructions that cause interrupts (other than trace interrupts)
- The first instruction of any interrupt handler
- Instructions that are emulated by software

MSR[SE,BE] are both cleared when the trace interrupt is taken. In the normal use of this function, MSR[SE,BE] are restored when the interrupt handler returns to the interrupted program using an **rfi** instruction.

Register settings for the trace mode are described in Table 6-19.

Table 6-19. Trace Interrupt—Register Settings

Register	Setting Description														
SRR0	Set to the EA of the next instruction to be executed in the program for which the trace interrupt was generated.														
SRR1	1–4, 10–15 Cleared 16–23, 25–27 Loaded with equivalent bits from the MSR 30–31 Note that depending on the implementation, additional MSR bits may be copied to SRR1.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

When a trace interrupt is taken, instruction execution resumes at offset 0x00D00 from the base address determined by MSR[IP].

6.5.12 Floating-Point Assist Interrupt (0x00E00)

The floating-point assist interrupt is optional to the PowerPC architecture. It can be used to allow software to assist in the following situations:

- Execution of floating-point instructions for which an implementation uses software routines to perform certain operations, such as those involving denormalization.
- Execution of floating-point instructions that are not optional and are not implemented in hardware. In this case, the processor may generate an illegal instruction type program interrupt instead.

Register settings for the floating-point assist interrupts are described in Table 6-20.

Table 6-20. Floating-Point Assist Interrupt—Register Settings

Register	Setting Description														
SRR0	Set to the EA of the next instruction to be executed in the program for which the floating-point assist interrupt was generated.														
SRR1	1–4, 10–15 Implementation-specific information 16–23, 25–27 Loaded with equivalent bits from the MSR 30–31 Note that depending on the implementation, additional MSR bits may be copied to SRR1.														
MSR	**POW**	**ILE**	**EE**	**PR**	**FP**	**ME**	**FE0**	**SE**	**BE**	**FE1**	**IP**	**IR**	**DR**	**RI**	**LE**
	0	—	0	0	0	—	0	0	0	0	—	0	0	0	Set to value of ILE

When a floating-point assist interrupt is taken, instruction execution resumes as offset 0x00E00 from the base address determined by MSR[IP].

Chapter 7
Memory Management

This chapter gives an overview of the memory management model, which defines how addresses are translated and how memory is protected. Note that some implementations that comply with the operating environments architecture (OEA) extend the architectural definitions described here, for example, through the implementation of a 36-bit physical address space and duplication of the block address translation registers (BATs). Although the specifics of those extensions are described in the implementation documentation, the information in this chapter provides a basic understanding of the memory management model that is used by all processors that implement the OEA.

7.1 Overview

Instruction and data accesses generated by load and store instructions require address translation. Also, the addresses specified by cache instructions and the optional external control instructions require translation. Generally, the address translation mechanism is defined in terms of the segment registers (SRs) and page tables used to locate the effective-to-physical address mapping for memory accesses. The segment information translates the effective address to an interim virtual address, and the page table information translates the virtual address to a physical address.

The definition of the page table data structures provides significant flexibility for the implementation of performance enhancement features in a wide range of processors. Therefore, the performance enhancements used to store page table information on-chip vary from implementation to implementation.

Translation lookaside buffers (TLBs) are commonly implemented to keep recently used page address translations on-chip. Although their exact characteristics are not specified in the OEA, the general concepts that are pertinent to the system software are described.

The segment information, used to generate the interim virtual addresses, is stored in the on-chip SRs.

The block address translation (BAT) mechanism is a software-controlled array that stores the available block address translations on-chip. BAT array entries are implemented as pairs of BAT registers that are accessible as supervisor special-purpose registers (SPRs).

The MMU and interrupt mechanism provide the necessary support for the operating system to implement a paged virtual memory environment and for enforcing protection of designated memory areas. Interrupt processing is described in Chapter 6, “Interrupts.” Section 2.3.1, “Machine State Register (MSR),” describes the MSR, which controls some of the critical MMU functionality.

7.2 MMU Features

The MMU provides 4 Gbytes of effective address space, a 52-bit interim virtual address, and physical addresses that are ≤ 32 bits in length. Note that this chapter describes address translation mechanisms from the perspective of the programming model. As such, it describes the structure of the page and segment tables, the MMU conditions that cause interrupts, the instructions provided for programming the MMU, and the MMU registers. The hardware implementation details of a particular MMU (including whether the hardware automatically performs a page table search in memory) are not contained in the architectural definition and are invisible to the programming model; therefore, they are not described in this document. In the case that some of the OEA model is implemented with some software assist mechanism, this software should be contained in the area of memory reserved for implementation-specific use and should not be visible to the operating system.

7.3 MMU Overview

This section is an overview of the high-level organization and operational concepts of the MMU and a summary of all MMU control registers. The MMU and interrupt models support demand-paged virtual memory, which permits execution of programs larger than the size of physical memory. The term ‘demand paged’ implies that individual pages are loaded into physical memory from backing storage only as they are accessed by an executing program.

The memory management model includes the concept of a virtual address that is not only larger than that of the maximum allowed physical memory but a virtual address space that is also larger than the effective address space. Effective addresses are 32 bits wide. In the address translation process, the processor converts an effective address to a 52-bit virtual address, as per the information in the selected descriptor. Then the address is translated back to a physical address the size (or less) of the effective address.

Note that in the cases that implementations support a physical address range that is smaller than 32 bits, the high-order bits of the effective address may be ignored in the address translation process. The remainder of this chapter assumes that implementations support the maximum physical address range.

The operating system manages the system’s physical memory resources. It initializes the MMU registers (SRs, BATs, and SDR1) and sets up page tables in memory. The MMU then assists the operating system by managing page status and optionally caching the recently-used address translation information on-chip for quick access.

Effective address spaces are divided into 256-Mbyte regions called segments or into other large regions called blocks (128 Kbyte–256 Mbyte). Segments that correspond to memory-mapped areas can be further subdivided into 4-Kbyte pages. For each block or page, the operating system creates an address descriptor (page table entry (PTE) or BAT array entry); the MMU then uses these descriptors to generate the physical address, the protection information, and other access control information each time an address within the block or page is accessed. Address descriptors for pages reside in tables (as PTEs) in physical memory; for faster accesses, the MMU often caches on-chip copies of recently-used PTEs in an on-chip TLB. The MMU keeps the block information on-chip in the BAT array (comprised of the BAT registers).

For more information, see Section 2.3.1, “Machine State Register (MSR),” Section 7.5.3, “BAT Register Implementation of BAT Array,” Section 7.6.2.1, “Segment Register Definitions,” and Section 7.7.1.1, “SDR1 Register Definitions.”

7.3.1 Memory Addressing

A program references memory using the effective (logical) address computed by the processor when it executes a load, store, branch, or cache instruction, and when it fetches the next instruction. The effective address is translated to a physical address according to the procedures described throughout this chapter. The memory subsystem uses the physical address for the access.

7.3.1.1 Predefined Physical Memory Locations

Four areas of the physical memory map have predefined uses. The first 256 bytes of physical memory (or if MSR[IP] = 1, the first 256 bytes of memory located at physical address 0xFFF0_0000) are assigned for arbitrary use by the operating system. The rest of that first page of physical memory defined by the vector base address (determined by MSR[IP]) is used for interrupt vectors or reserved for future interrupt vectors. The third area consists of the second and third physical pages of the memory map, which are used for implementation-specific purposes. In some implementations, these pages are located at physical address 0xFFF0_1000 when MSR[IP] = 1 are also used for implementation-specific purposes. The system software defines the fourth area for locations in physical memory that contain the page address translation tables. These predefined memory areas are summarized in Table 7-1.

Table 7-1. Predefined Physical Memory Locations

Memory Area	Physical Address Range	Predefined Use
1	Base[1] || 0x0_0000–Base || 0x0_00FF	Operating system
2	Base[1] || 0x0_0100–Base || 0x0_0FFF	Interrupt vectors
3	Base[1] || 0x0_1000–Base || 0x0_2FFF	Implementation-specific· Valid only for MSR[IP] = 1 on some implementations.
4	Software-specified—contiguous sequence of physical pages	Page table

[1] MSR[IP] = 0, base = 0x000; MSR[IP] = 1, base = 0xFFF

Chapter 6, “Interrupts,” lists the assignments of the interrupt vector offsets.

7.3.2 MMU Organization

Figure 7-1 shows a conceptual block diagram of the MMU. The low-order address bits, A20–A31 are untranslated and therefore identical for both effective and physical addresses. After translating the address, the MMU passes the resulting 32-bit physical address to the memory subsystem.

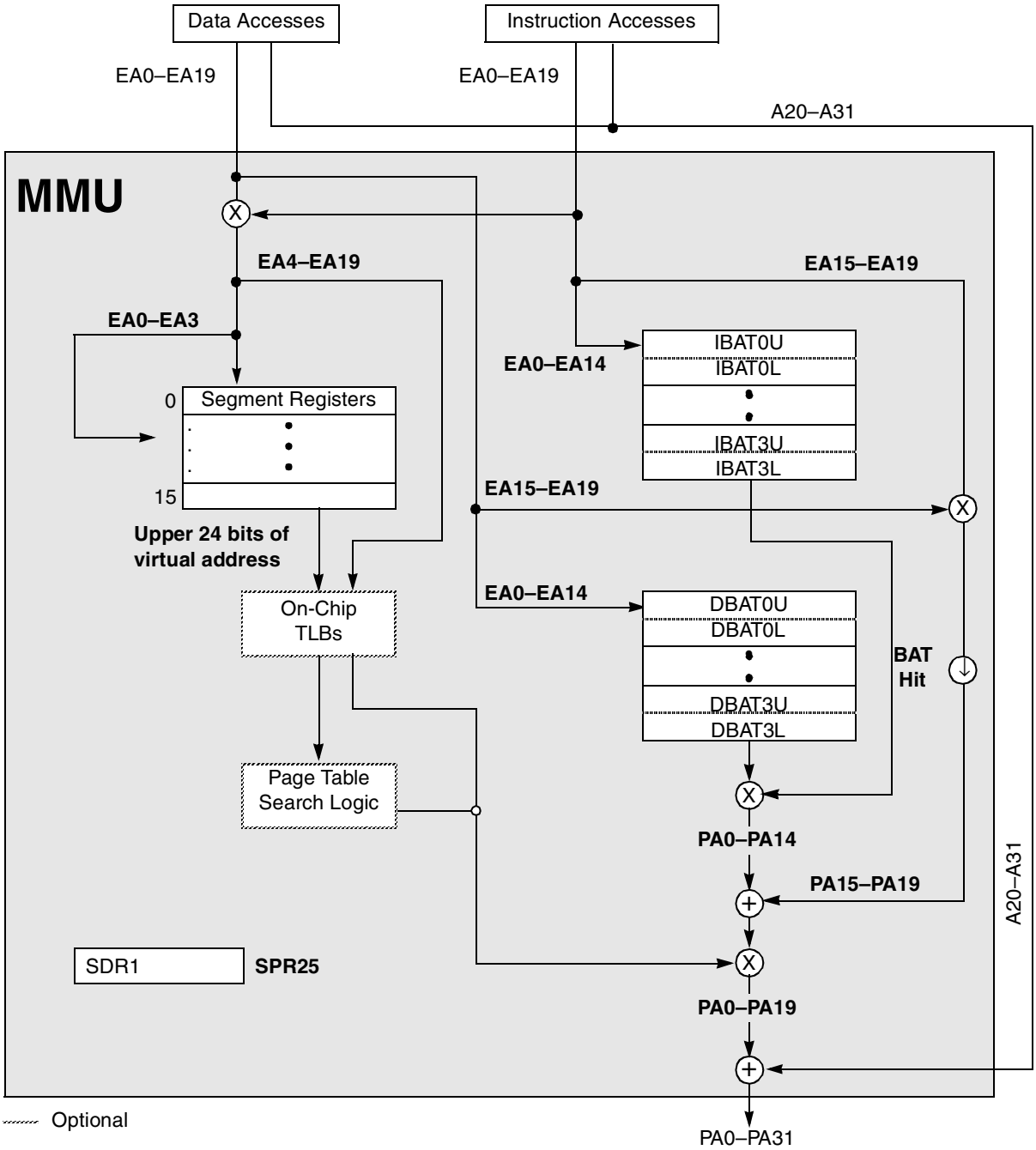


Figure 7-1. MMU Conceptual Block Diagram

7.3.3 Address Translation Mechanisms

The following types of address translation are supported:

- Page address translation—translates the page frame address for a 4-Kbyte page size
- Block address translation—translates the block number for blocks that range from 128 Kbytes to 256 Mbytes
- Real addressing mode address translation—when address translation is disabled, the physical address is identical to the effective address.

In addition, earlier processors implement a direct-store facility that is used to generate direct-store interface accesses on the external bus. Note that this facility is not optimized for performance and was present for compatibility with POWER devices. Future devices are not likely to support it; software should not depend on its effects and new software should not use it.

Figure 7-2 shows the address translation mechanisms provided by the MMU. As Figure 7-2 shows, when an access uses the page or direct-store segment address translation, the appropriate SR is selected by the highest-order effective address bits.

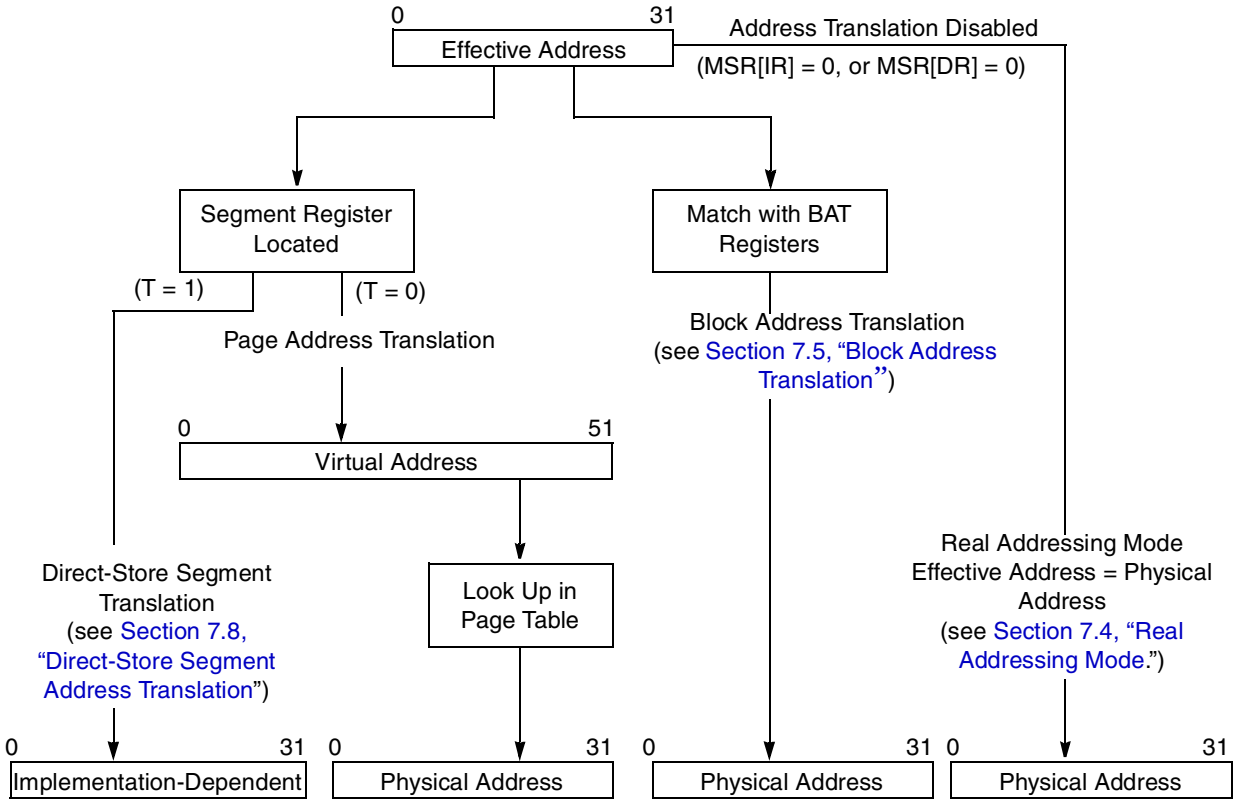


Figure 7-2. Address Translation Types

The corresponding SR*n*[T] bit determines whether the access is to memory (memory-mapped) or to a direct-store segment. Note that the direct-store interface is present to allow certain older I/O devices to use this interface. When an access is determined to be to the direct-store interface space, the implementation invokes an elaborate hardware protocol for communication with these devices. The direct-store interface

protocol is not optimized for performance, and therefore, its use is discouraged. The most efficient method for accessing I/O is by memory-mapping the I/O areas.

For memory accesses translated by an SR, the interim virtual address is generated using the information in the SR. Page address translation corresponds to the conversion of this virtual address into the 32-bit physical address used by the memory subsystem. In some cases, the physical address for the page is available in an on-chip TLB for quick access. However, if the page address translation misses in a TLB, the MMU searches the page table in memory (using the virtual address information and a hashing function) to locate the required physical address. In some implementations, dedicated hardware performs the page table search automatically; in others, an interrupt handler routine searches the page table with software.

Because blocks are larger than pages, there are fewer upper-order effective address bits to be translated into physical address bits (more low-order address bits (at least 17) are untranslated to form the offset into a block) for block address translation. Also, instead of SRs and a page table, block address translations use the on-chip BAT registers as a BAT array. If an effective address matches the corresponding BAT register field, the information in the BAT register is used to generate the physical address; in this case, the results of the page translation (occurring in parallel) are ignored. Note that a matching BAT array entry always takes precedence over a translation provided by the SR (even if the segment is a direct-store segment).

Direct-store address translation is used when the optional direct-store translation control bit (SR*n*[T]) is set. In this case, the remaining information in the SR is interpreted as identifier information that is used with the remaining effective address bits to generate the protocol used in a direct-store interface access on the external interface; additionally, no TLB lookup or page table search is performed. Note that this facility is not likely to be supported in future processors.

When the processor generates an access and the corresponding address translation enable bit in MSR is zero, the resulting physical address is identical to the effective address and all other translation mechanisms are ignored. Instruction and data address translation is enabled by setting MSR[IR] and MSR[DR]. See Section 7.3.6.1, “Real Addressing Mode and Block Address Translation Selection.”

7.3.4 Memory Protection Facilities

In addition to the translation of effective addresses to physical addresses, the MMU provides access protection of supervisor areas from user access and can designate areas of memory as read-only as well as no-execute. Figure 7-2 shows the eight protection options supported by the MMU for pages.

Table 7-2. Access Protection Options for Pages

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Supervisor-only	—	—	—	√	√	√
Supervisor-only-no-execute	—	—	—	—	√	√
Supervisor-write-only	√	√	—	√	√	√
Supervisor-write-only-no-execute	—	√	—	—	√	√
Both user/supervisor	√	√	√	√	√	√
Both (user/supervisor)-no-execute	—	√	√	—	√	√

Table 7-2. Access Protection Options for Pages (continued)

Option	User Read		User Write	Supervisor Read		Supervisor Write
	I-Fetch	Data		I-Fetch	Data	
Both (user/supervisor) read-only	√	√	—	√	√	—
Both (user/supervisor) read-only-no-execute	—	√	—	—	√	—

√ Access permitted
— Protection violation

The no-execute option provided in the SR lets the operating system program whether or not instructions can be fetched from an area of memory. The remaining options are enforced based on a combination of information in the SR and the page table entry. Thus, the supervisor-only option allows only read and write operations generated while the processor is operating in supervisor mode (MSR[PR] = 0) to access the page. User accesses that map into a supervisor-only page cause an interrupt.

Note that independently of the protection mechanisms, care must be taken when writing to instruction areas as coherency must be maintained with on-chip copies of instructions that may have been prefetched into a queue or an instruction cache. Section 5.2.5.2, “Instruction Cache Instructions,” describes coherency within instruction areas.

As shown in the table, the supervisor write-only option allows both user and supervisor accesses to read from the page, but only supervisor programs can write to that area. There is also an option that allows both supervisor and user programs read and write access (both user/supervisor option), and finally, there is an option to designate a page as read-only, both for user and supervisor programs (both read-only option).

For areas of memory that are translated by the block address translation mechanism, the protection options are similar, except that blocks are translated by separate mechanisms for instruction and data, blocks do not have a no-execute option, and blocks can be designated as enabled for user and supervisor accesses independently. Therefore, a block can be designated as supervisor-only, for example, but this block can be programmed such that all user accesses simply ignore the block translation, rather than take an interrupt in the case of a match. This allows a flexible way for supervisor and user programs to use overlapping effective address space areas that map to unique physical address areas (without interrupts occurring).

For direct-store segments, the MMU calculates a key bit based on the protection values programmed in the SR and the specific user/supervisor and read/write information for the access. However, this bit is merely passed on to the system interface to be transmitted in the context of the direct-store interface protocol. The MMU does not itself enforce any protection or cause any interrupt based on the state of the key bit for such accesses. The I/O controller device or other external hardware can optionally use this bit to enforce any protection required. Note that future devices are unlikely to implement the direct-store facility.

Finally, a facility in the VEA and OEA allows pages or blocks to be designated as guarded, preventing speculative (referred to as out-of-order in the architecture) accesses that may cause undesired side effects. For example, areas of the memory map used to control I/O devices can be marked as guarded so accesses do not occur unless the program explicitly requires them. Note that the terms ‘speculative’ and ‘out-of-order’ are used here as defined in Section 5.3.1.5.1, “Definition of Speculative and Out-of-Order Memory Accesses.” Refer to Section 5.3.1.5.4, “Speculative Accesses to Guarded Memory,” for a complete description of how accesses to guarded memory are restricted.

7.3.5 Page History Information

The MMU model also defines reference (R) and change (C) bits in the page address translation mechanism that can be used as history information relevant to the page. The operating system can use these bits to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Although these bits are initially programmed by the operating system into the page table, the architecture specifies that the processor maintains the R and C bits and updates those bits when required.

7.3.6 General Flow of MMU Address Translation

The following sections describe the general flow used to translate effective addresses to virtual and then physical addresses. Note that although there are references to the concept of an on-chip TLB, these entities may not be present in a particular hardware implementation for performance enhancement (and an implementation may have one or more TLBs). Thus, they are shown here as optional and only the software ramifications of the existence of a TLB are discussed.

7.3.6.1 Real Addressing Mode and Block Address Translation Selection

If an instruction or data access is generated and the corresponding translation is disabled (MSR[IR] = 0 or MSR[DR] = 0), real addressing mode translation is used (physical address equals effective address) and the access continues to the memory subsystem as described in Section 7.4, “Real Addressing Mode.”

Figure 7-3 shows the flow the MMU uses in determining whether to select real addressing mode, block address translation, or the SR (to select either direct-store or page address translation).

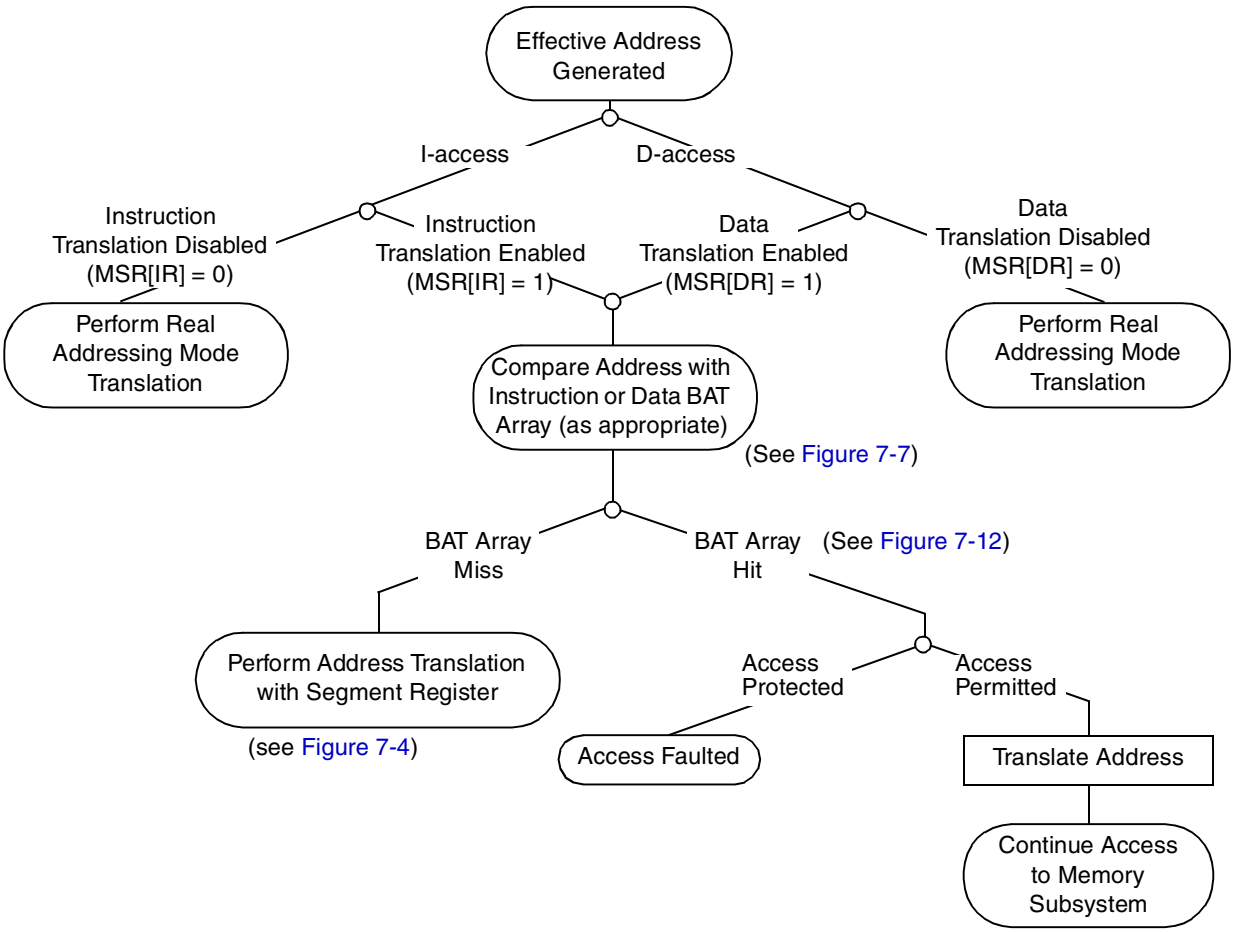


Figure 7-3. General Flow of Address Translation (Real Addressing Mode and Block)

Note that if the BAT array search results in a hit, the access is qualified with the appropriate protection bits. If the access is determined to be protected (not allowed), an instruction storage interrupt (ISI) or data storage interrupt (DSI) is generated.

7.3.6.2 Page and Direct-Store Address Translation Selection

If address translation is enabled and the effective address information does not match a BAT array entry, the SR must be located. EA0–EA3 select one of the 16 SRs. As shown in Figure 7-4, SR[T] selects whether the translation is to a page or to a direct-store segment. Figure 7-4 also shows how the no-execute protection is enforced; if SR*n*[N] is set and the access is an instruction fetch, the access is faulted.

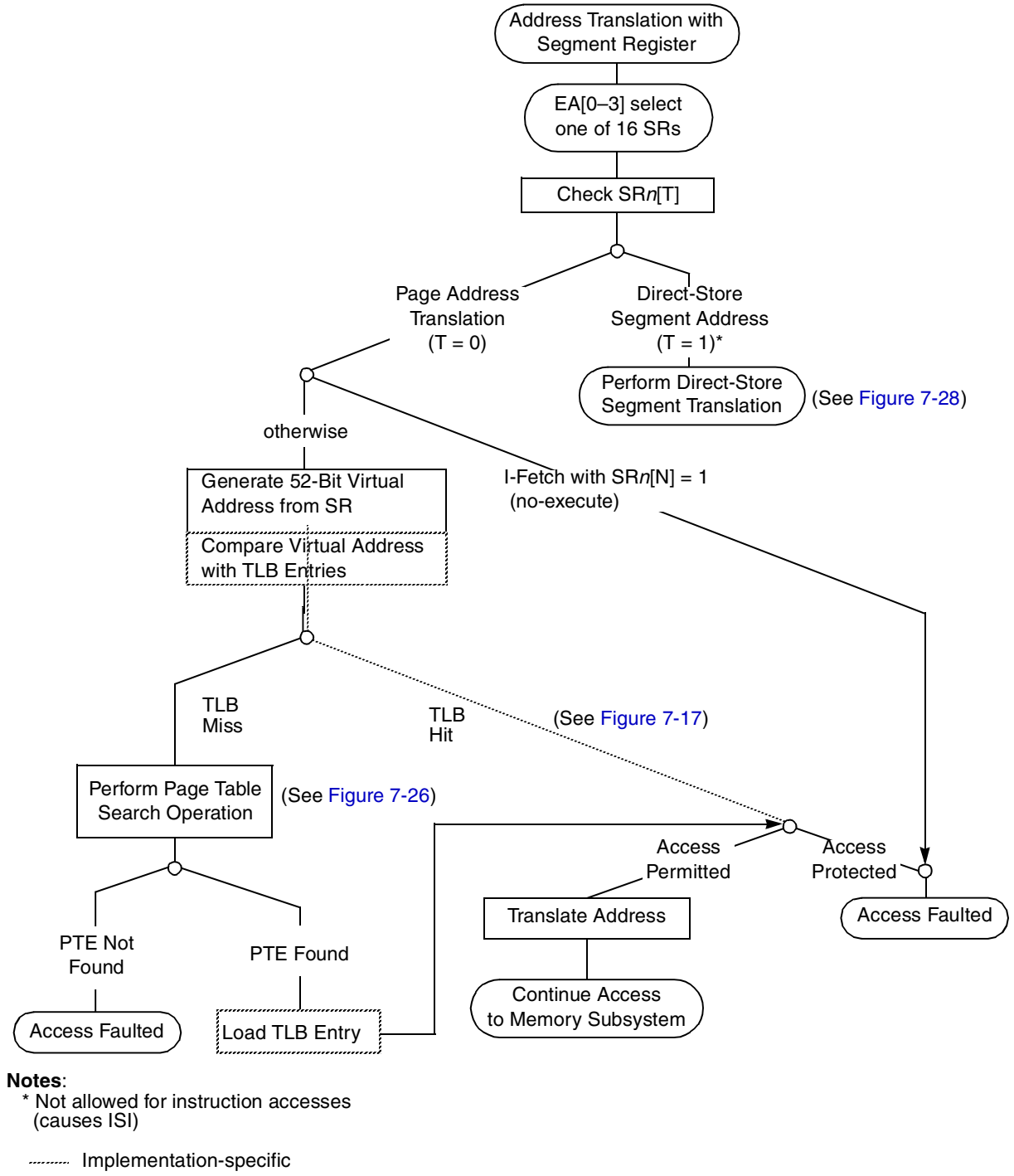


Figure 7-4. General Flow of Page and Direct-Store Address Translation

7.3.6.2.1 Selection of Page Address Translation

If SR[T] = 0, page address translation is selected. The information in the SR is then used to generate the 52-bit virtual address. The virtual address is then used to identify the page address translation information (stored as page table entries (PTEs) in a page table in memory). Once again, although the architecture does

not require the existence of a TLB, one or more TLBs may be implemented in the hardware to store copies of recently-used PTEs on-chip for increased performance.

If an access hits in the TLB, the page translation occurs and the physical address bits are forwarded to the memory subsystem. If the translation is not found in the TLB, the MMU requires a search of the page table. The hardware of some implementations may perform the table search automatically, while others may trap to an interrupt handler for the system software to perform the page table search. If the translation is found, a new TLB entry is created and the page translation is once again attempted. This time, the TLB is guaranteed to hit. When the PTE is located, the access is qualified with the appropriate protection bits. If the access is determined to be protected (not allowed), an interrupt (ISI or DSI) is generated.

If the PTE is not found by the table search operation, an ISI or DSI is generated.

7.3.7 MMU Interrupts Summary

To complete any memory access, the effective address must be translated to a physical address. A translation exception occurs if this translation fails for one of the following reasons:

- There is no valid entry in the page table for the page specified by the effective address (and SR) and there is no valid BAT translation.
- There is no valid SR and there is no valid BAT translation.
- An address translation is found but the access is not allowed by the memory protection mechanism.

The translation exceptions cause either the instruction storage interrupt (ISI) or the data storage interrupt (DSI) to be taken as shown in Figure 7-3. The state saved by the processor for each of these interrupts contains information that identifies the address of the failing instruction. Refer to Chapter 6, “Interrupts,” for a more detailed description of interrupt processing, and the SRR1 and DSISR bit settings when an interrupt occurs.

Table 7-3. Translation Exceptions

Exception	Description	Interrupt
Page fault (no PTE found)	No matching PTE found in page tables (and no matching BAT array entry)	I access: ISI SRR1[1] = 1
		D access: DSI DSISR[1] = 1
Block protection violation	Conditions described in Table 7-9 for block	I access: ISI SRR1[4] = 1
		D access: DSI DSISR[4] = 1
Page protection violation	Conditions described in Table 7-16 for page	I access: ISI SRR1[4] = 1
		D access: DSI DSISR[4] = 1
No-execute protection violation	Attempt to fetch instruction when SR[N] = 1	ISI SRR1[3] = 1

Table 7-3. Translation Exceptions (continued)

Exception	Description	Interrupt
Instruction fetch from direct-store segment. Note that the optional direct-store is being removed from the architecture.	Attempt to fetch instruction when SR[T] = 1	ISI SRR1[3] = 1
Instruction fetch from guarded memory	Attempt to fetch instruction when MSR[IR] = 1 and either: matching xBAT[G] = 1 or no matching BAT entry and PTE[G] = 1	ISI SRR1[3] = 1

In addition to the translation exceptions, other MMU-related conditions (some implementation-specific) can cause an interrupt. Table 7-4 shows how these exceptions map to interrupts. If MSR[DR] = 0, the only MMU exceptions that occur are those that cause the alignment interrupt for data accesses. For detailed information about exceptions that cause the alignment interrupt (in particular for string/multiple instructions), see Section 6.5.6, "Alignment Interrupt (0x00600)." Chapter 6, "Interrupts," describes SRR1 and DSISR bit settings for these interrupts.

Table 7-4. Other MMU Exceptions

Condition	Description	Interrupt
dcbz with W = 1 or I = 1 (may cause interrupt or operation may be performed to memory)	**dcbz** instruction to write-through or cache-inhibited segment or block	Alignment interrupt (implementation-dependent)
lwarx or **stwcx.** with W = 1 (may cause interrupt or execute correctly)	Reservation instruction to write-through segment or block	DSI (implementation-dependent) DSISR[5] = 1
lwarx, **stwcx.**, **eciwx**, or **ecowx** to a direct-store[1] segment (may cause interrupt or produce boundedly-undefined results).	Reservation instruction or external control instruction when SR[T] = 1	DSI (implementation-dependent) DSISR[5] = 1
Floating-point load or store to direct-store[1] segment (instruction may execute correctly or cause interrupt)	Floating-point memory access when SR[T] = 1	Alignment interrupt (implementation-dependent)
Load or store operation that causes a direct-store[1] error.	Direct-store interface protocol signalled with an error condition	DSI DSISR[0] = 1
eciwx or **ecowx** attempted when external control facility disabled	**eciwx** or **ecowx** attempted with EAR[E] = 0	DSI DSISR[11] = 1
lmw, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** attempted in little-endian mode	**lmw**, **stmw**, **lswi**, **lswx**, **stswi**, or **stswx** attempted while MSR[LE] = 1	Alignment interrupt
Operand misalignment	Translation enabled and operand is misaligned as described in Chapter 6, "Interrupts."	Alignment interrupt (some cases are implementation-dependent)

[1] The direct-store facility is optional and being removed from the architecture.

7.3.8 MMU Instructions and Register Summary

The MMU instructions and registers allow the operating system to set up the SRs. Additionally, the operating system has the resources to set up the block address translation areas and the page tables in memory.

Note that because the implementation of TLBs is optional, instructions that refer to TLBs are also optional. However, because TLBs serve as caches of the page table, there must be a software protocol for maintaining coherency between these caches and the tables in memory whenever the tables in memory are modified. Therefore, the OEA specifies that a processor implementing a TLB is guaranteed to have a means for doing the following:

- Invalidating an individual TLB entry
- Invalidating the entire TLB

When the tables in memory are changed, the operating system purges these caches of the corresponding entries, allowing the translation caching mechanism to refetch from the tables when the corresponding entries are required.

A processor may implement one or more of the instructions described in this section to support table invalidation. Alternatively, an algorithm may be specified that performs one of the functions listed above (a loop invalidating individual TLB entries may be used to invalidate the entire TLB, for example), or different instructions may be provided.

A processor may also perform additional functions (not described here) as well as those described in the implementation of some of these instructions. For example, the **tlbie** instruction may be implemented so as to purge all TLB entries in a congruence class (that is, all TLB entries indexed by the specified EA which can include corresponding entries in data and instruction TLBs) or the entire TLB.

Note that if a processor does not implement an optional instruction it treats the instruction as a no-op or as an illegal instruction, depending on the implementation. Also, note that the segment register and TLB concepts described here are conceptual; that is, a processor may implement parallel sets of segment registers (and even TLBs) for instructions and data.

Because the MMU specification is so flexible, it is recommended that the software that uses these instructions and registers be encapsulated into subroutines to minimize the impact of migrating across the family of implementations.

Table 7-5 summarizes the instructions that specifically control the MMU. For more detailed information about the instructions, refer to Chapter 8, “Instruction Set.”

Table 7-5. Instruction Summary—Control MMU

Instruction	Syntax	Description
Move to Segment Register	**mtsr** SR,**r**S	SR[SR]← **r**S
Move to Segment Register Indirect	**mtsrin r**S,**r**B	SR[**r**B[0–3]]←**r**S
Move from Segment Register	**mfsr r**D,SR	**r**D←SR[SR]
Move from Segment Register Indirect	**mfsrin r**D,**r**B	**r**D←SR[**r**B[0–3]]
Translation Lookaside Buffer Invalidate All (optional)	**tlbia**	For all TLB entries, TLB[V]←0 Invalidates TLB entries only for processor that executed **tlbia**
Translation Lookaside Buffer Invalidate Entry (optional)	**tlbie r**B	If TLB hit (for effective address specified as **r**B), TLB[V]←0 Causes TLB invalidation of entry in all processors in system
Translation Lookaside Buffer Synchronize (optional)	**tlbsync**	Ensures that all **tlbie** instructions previously executed by the processor executing **tlbsync** have completed on all processors

Figure 7-5 shows the registers that the operating system uses to program the MMU. These registers are accessible to supervisor-level software only (supervisor level is referred to as privileged state in the architecture specification).

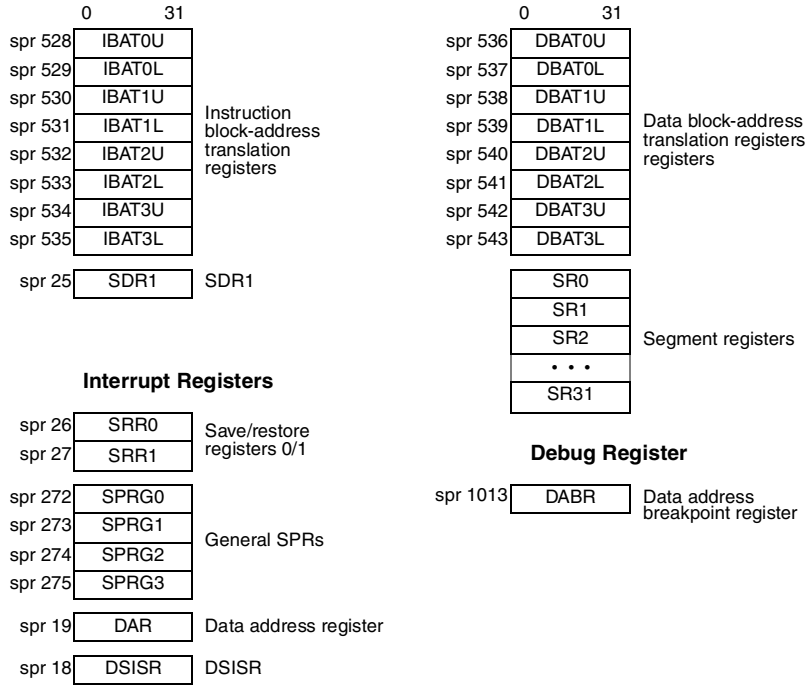


Figure 7-5. MMU Registers

Table 7-6 summarizes these registers, which are described in detail in Section 2.3, “OEA Register Set.”

Table 7-6. MMU Registers

Register	Description
Segment registers (SR0–SR15)	Figure 7-14 shows the format of a segment register. SR fields in are interpreted differently depending on the value of bit 0. SRs are accessed by the **mtsr**, **mtsrin**, **mfsr**, and **mfsrin** instructions. See Section 2.3.5, “Segment Registers.”
BAT registers (IBAT0U–IBAT3U, IBAT0L–IBAT3L, DBAT0U–DBAT3U, DBAT0L–DBAT3L)	The 16 BAT registers are organized as four pairs of instruction BAT SPRs (IBAT0U–IBAT3U paired with IBAT0L–IBAT3L) and four pairs of data BAT SPRs (DBAT0U–DBAT3U paired with DBAT0L–DBAT3L). See Section 2.3.3, “BAT Registers.”
SDR1 register	Specifies the base and size of the page tables in memory. See Section 2.3.4, “SDR1.”

7.3.9 TLB Entry Invalidation

Optionally, processors implement TLB structures that store on-chip copies of those PTEs that are resident in physical memory. Such processors can use **tlbie** and **tlbia** instructions to invalidate resident TLB entries. These instructions may also enable a TLB invalidate signalling mechanism in hardware so that other processors also invalidate their resident copies of the matching PTE. See Chapter 8, “Instruction Set,” for detailed information about **tlbie** and **tlbia**.

7.4 Real Addressing Mode

If address translation is disabled (MSR[IR] = 0 or MSR[DR] = 0) for a particular access, the effective address is treated as the physical address and is passed directly to the memory subsystem as a real addressing mode address translation. If an implementation has a smaller physical address range than effective address range, the extra high-order bits of the effective address may be ignored in the generation of the physical address.

Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” describes synchronization requirements for changes to MSR[IR] and MSR[DR].

The addresses for accesses that occur in real addressing mode bypass all memory protection checks as described in Section 7.5.4, “Block Memory Protection,” and Section 7.6.4, “Page Memory Protection” and do not cause the recording of reference and change information (described in Section 7.6.3, “Page History Recording”).

For data accesses that use real addressing mode, the memory access mode bits (WIMG) are assumed to be 0b0011. That is, the cache is write-back and memory does not need to be updated immediately (W = 0), caching is enabled (I = 0), data coherency is enforced with memory, I/O, and other processors (caches) (M = 1, so data is global), and the memory is guarded. For instruction accesses in real addressing mode, the memory access mode bits (WIMG) are assumed to be either 0b0001 or 0b0011. That is, caching is enabled (I = 0) and the memory is guarded. Additionally, coherency may or may not be enforced with memory, I/O, and other processors (caches) (M = 0 or 1, so data may or may not be considered global). For a complete description of the WIMG bits, refer to Section 5.3.1, “Memory/Cache Access Attributes.”

Note that attempting to execute **eciwx** or **ecowx** while MSR[DR] = 0 causes boundedly-undefined results.

Whenever an interrupt occurs, the processor clears both MSR[IR] and MSR[DR]. Therefore, at least at the beginning of all interrupt handlers (including reset), the processor operates in real addressing mode for instruction and data accesses. If address translation is required for the interrupt handler code, the software must explicitly enable address translation by accessing the MSR as described in Section 2.3.1, “Machine State Register (MSR).”

Note that an attempt to access a physical address that is not physically present in the system may cause a machine check interrupt (or even a checkstop condition), depending on the response by the system for this case. Thus, care must be taken when generating addresses in real addressing mode. Note that this can also occur when translation is enabled and SDR1 sets up the translation such that nonexistent memory is accessed. For more information, see Section 6.5.2, “Machine Check Interrupt (0x00200).”

7.5 Block Address Translation

The block address translation (BAT) mechanism in the OEA provides a way to map ranges of effective addresses larger than a single page into contiguous areas of physical memory. Such areas can be used for data that is not subject to normal virtual memory handling (paging), such as a memory-mapped display buffer or an extremely large array of numbers.

The following sections describe the implementation of block address translation, including the block protection mechanism, followed by a block translation summary with a detailed flow diagram.

7.5.1 BAT Array Organization

The block address translation mechanism is implemented as a software-controlled BAT array, which maintains the address translation information for eight blocks of memory. The BAT array is maintained by the system software and is implemented as a set of 16 supervisor-level SPRs. Each block is defined by a pair of SPRs called upper and lower BAT registers, which contain the effective and physical addresses for the block.

Section 7.5.3, “BAT Register Implementation of BAT Array,” gives more information about the BAT registers. Note that the BAT array entries are ignored for TLB invalidate operations detected in hardware and in the execution of the **tlbie** or **tlbia** instruction.

Figure 7-6 shows the organization of the BAT array. Four pairs of BAT registers are provided for translating instruction addresses and four pairs of BAT registers are used for translating data addresses. These eight pairs of BAT registers comprise two four-entry fully-associative BAT arrays (each BAT array entry corresponds to a pair of BAT registers). The BAT array is fully-associative in that any address can reside in any BAT. In addition, the effective address field of all four corresponding entries (instruction or data) is simultaneously compared with the effective address of the access to check for a match.

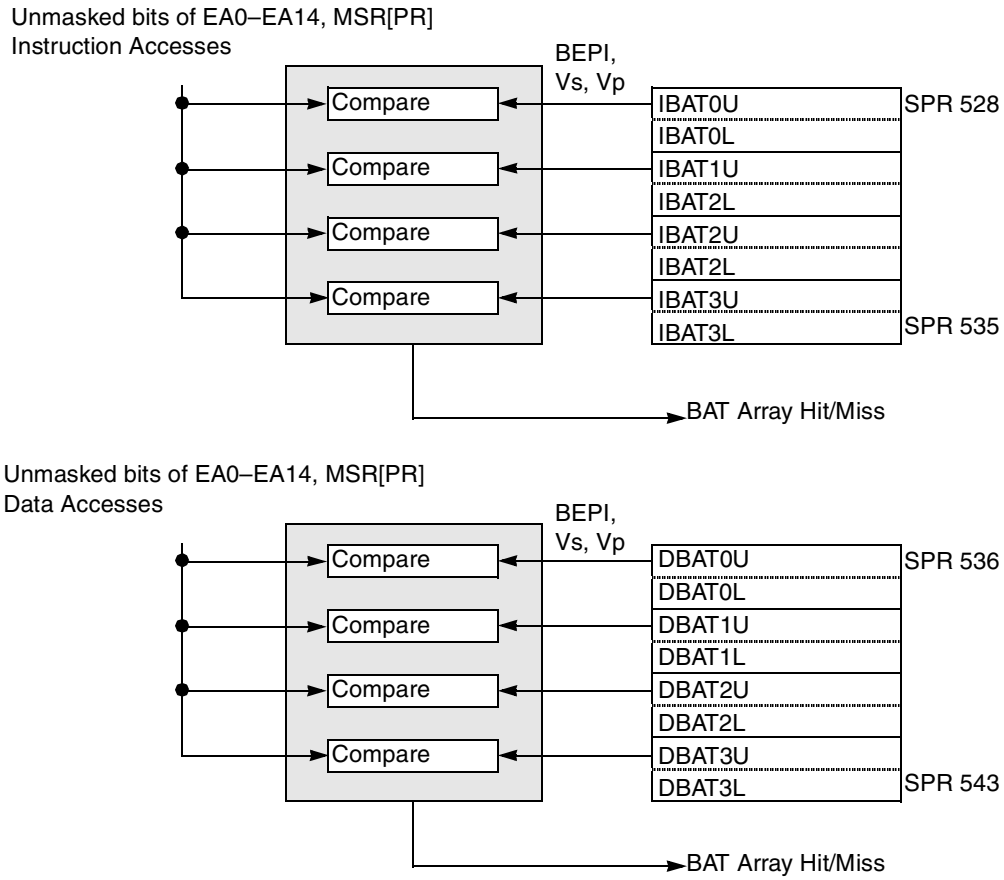


Figure 7-6. BAT Array Organization

Each pair of BAT registers defines the starting address of a block in the effective address space, the size of the block, and the start of the corresponding block in physical address space. If an effective address is within the range defined by a pair of BAT registers, its physical address is defined as the starting physical address of the block plus the low-order effective address bits.

Blocks are restricted to a finite set of sizes, from 128 Kbytes (2^{17} bytes) to 256 Mbytes (2^{28} bytes). The starting address of a block in both effective address space and physical address space is defined as a multiple of the block size.

It is an error for system software to program the BAT registers such that an effective address is translated by more than one valid IBAT pair or more than one valid DBAT pair. If this occurs, the results are undefined and may include a spurious violation of the memory protection mechanism, a machine check interrupt, or a checkstop condition.

The following equation determines whether a BAT entry is valid for a particular access:

$$BAT\_entry\_valid = (Vs \ \& \ \neg MSR[PR]) \mid (Vp \ \& \ MSR[PR])$$

If a BAT entry is not valid for a given access, it does not participate in address translation for that access. Two BAT entries may not map an overlapping effective address range and be valid at the same time.

Entries that have complementary settings of V[s] and V[p] may map overlapping effective address blocks. Complementary settings would be as follows:

BAT entry A: Vs = 1, Vp = 0

BAT entry B: Vs = 0, Vp = 1

7.5.2 Recognition of Addresses in BAT Arrays

The BAT arrays are accessed in parallel with segmented address translation to determine whether a particular effective address corresponds to a block defined by the BAT arrays. If an effective address is within a valid BAT area, the physical address for the memory access is determined as described in Section 7.5.5, “Block Physical Address Generation.”

Block address translation is enabled only when address translation is enabled (MSR[IR] = 1 and/or MSR[DR] = 1). Also, a matching BAT array entry always takes precedence over any SR translation, independent of the setting of SR[T], and the SR information is ignored.

Figure 7-7 shows the flow of the BAT array comparison used in block address translation. When an instruction fetch operation is required, the effective address is compared with the four instruction BAT array entries; similarly, the effective addresses of data accesses are compared with the four data BAT array entries. The BAT arrays are fully-associative in that any of the four instruction or data BAT array entries can contain a matching entry (for an instruction or data access, respectively).

Note that Figure 7-7 assumes that the protection bits, BATL[PP], allow an access to occur. If not, an interrupt is generated, as described in Section 7.5.4, “Block Memory Protection.”

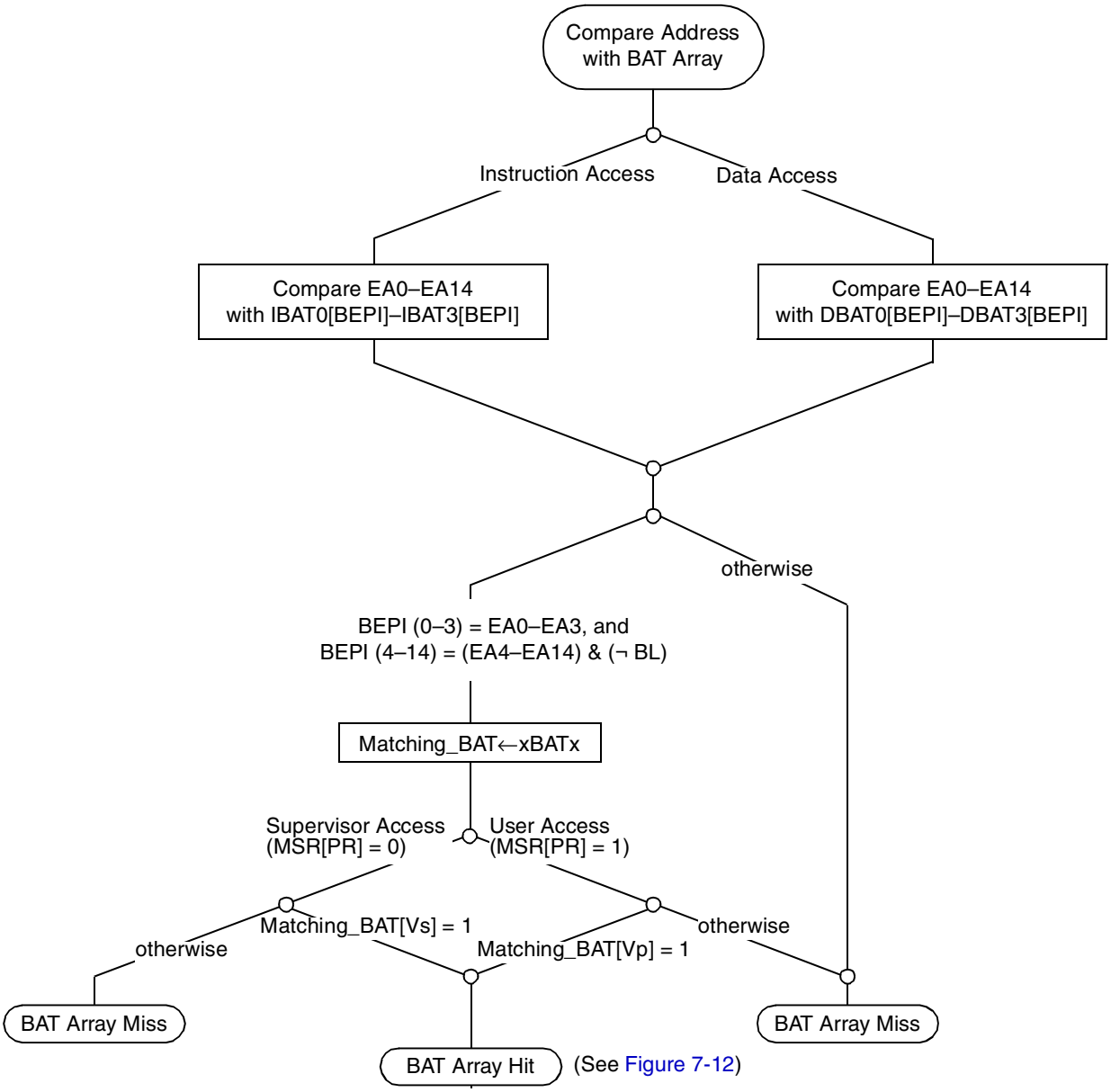


Figure 7-7. BAT Array Hit/Miss Flow

Two BAT array entry fields are compared to determine if there is a BAT array hit—a block effective page index (BEPI) field, which is compared with the high-order effective address bits, and one of two valid bits (Vs or Vp), which is evaluated relative to the value of MSR[PR]. Note that the figure assumes a block size of 128 Kbytes (all bits of BEPI are used in the comparison); the actual number of bits of the BEPI field that are used are masked by the BL field (block length) as described in Section 7.5.3, “BAT Register Implementation of BAT Array.”

Thus, the specific criteria for determining a BAT array hit are as follows:

- The upper-order 15 bits of the effective address, subject to a mask, must match the BEPI field of the BAT array entry.
- The appropriate valid bit in the BAT array entry must set to one as follows:
  - MSR[PR] = 0 corresponds to supervisor mode; in this mode, Vs is checked.
  - MSR[PR] = 1 corresponds to user mode; in this mode, Vp is checked.

The matching entry is then subject to the protection checking described in Section 7.5.4, “Block Memory Protection,” before it is used as the source for the physical address. Note that if a user mode program performs an access with an effective address that matches the BEPI field of a BAT area defined as valid only for supervisor accesses (Vp = 0 and Vs = 1) for example, the BAT mechanism does not generate a protection violation and the BAT entry is simply ignored. Thus, a supervisor program can use the block address translation mechanism to share a portion of the effective address space with a user program (that uses page address translation for this area).

If a memory area is to be mapped by the BAT mechanism for both instruction and data accesses, the mapping must be set up in both an IBAT and DBAT entry; this is the case even on implementations that do not have separate instruction and data caches.

Note that a block can be defined to overlay part of a segment such that the block portion is nonpaged although the rest of the segment can be paged. This allows nonpaged areas to be specified within a segment. Thus, if an area of memory is translated by an instruction BAT entry and data accesses are not also required to that same area of memory, PTEs are not required for that area of memory. Similarly, if an area of memory is translated by a data BAT entry, and instruction accesses are not also required to that same area of memory, PTEs are not required for that area of memory.

7.5.3 BAT Register Implementation of BAT Array

Recall that the BAT array is comprised of four entries used for instruction accesses and four entries used for data accesses. Each BAT array entry consists of a pair of BAT registers—an upper and a lower BAT register for each entry. The BAT registers are accessed with the **mtspr** and **mfspr** instructions and are only accessible to supervisor-level programs. See Appendix E, “Simplified Mnemonics for PowerPC Instructions,” for a list of simplified mnemonics for use with the BAT registers. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)

Figure 7-8 and Figure 7-9 show the formats and bit definitions of the upper and lower BAT registers.

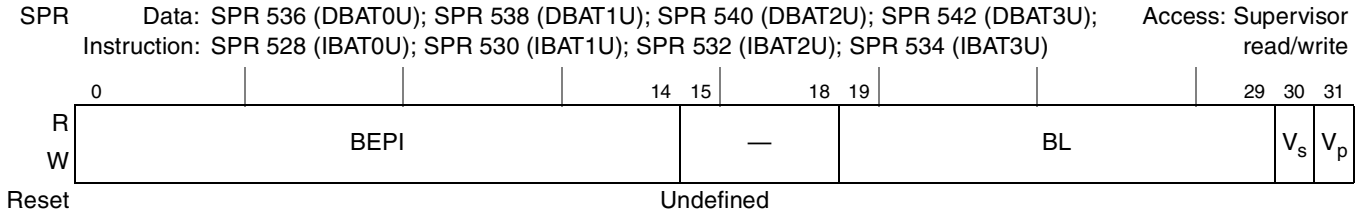
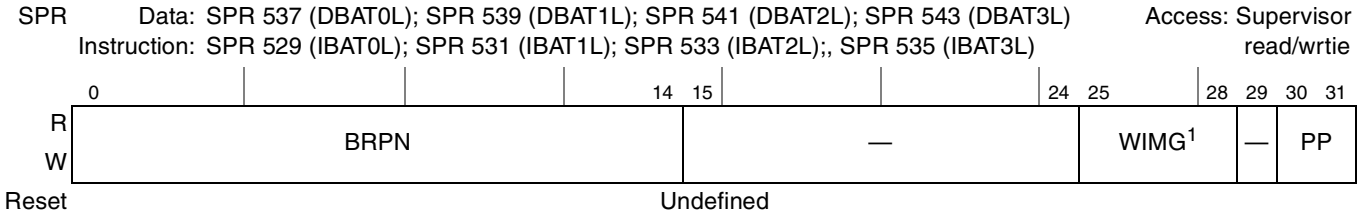


Figure 7-8. Format of Upper BAT Register



1 W and G bits are not defined for IBAT registers. Attempting to write to these bits causes boundedly-undefined results.

Figure 7-9. Format of Lower BAT Register

The BAT registers contain the effective-to-physical address mappings for blocks of memory. This mapping includes the memory/cache access mode bits (WIMG), the protection bits for the block, and effective address bits that are compared with the effective address of the access. The size of the block and the starting address of the block are defined by the physical block number (BRPN) and block size mask (BL) fields.

Figure 7-7 describes the bits in the upper and lower BAT registers. Note that the W and G bits are defined for BAT registers that translate data accesses (DBAT registers); attempting to write to the W and G bits in IBAT registers causes boundedly-undefined results.

Table 7-7. BAT Registers—Field and Bit Descriptions

Upper/Lower BAT	Bits	Name	Description
Upper BAT Register	0–14	BEPI	Block effective page index. Compared with high-order logical address bits to determine if there is a hit in that BAT array entry.
	15–18	—	Reserved
	19–29	BL	Block length. A mask that encodes block size. An EA lies within a BAT area if the appropriate EA bits (determined by BL) match BATU[BEPI] and if the appropriate valid bit (Vs or Vp) is set. Note that for an access to occur, the protection bits (PP) in the lower BAT register must be set appropriately, as described in Section 7.5.4, “Block Memory Protection.” The number of zeros in BL determines the EA bits used in the comparison with the BEPI field to determine if there is a hit in that BAT array entry. The rightmost BL bit is aligned with EA bit 14; EA bits corresponding to 1 bits in the BL field are then cleared for the comparison. The value loaded into BL determines both the block’s size and its alignment in both effective and physical address space. The BEPI and BRPN values must have at least as many low-order zeros as there are ones in BL. Otherwise, results are undefined. Also, if the processor does not support 32 bits of physical address, software should write zeros to those unsupported bits in BRPN (as the implementation treats them as reserved). Otherwise, a machine check interrupt can occur. Only the following BL values are valid: 000 0000 0000 128 Kbytes 000 0011 1111 8 Mbytes 000 0000 0001 256 Kbytes 000 0111 1111 16 Mbytes 000 0000 0011 512 Kbytes 000 1111 1111 32 Mbytes 000 0000 0111 1 Mbyte 001 1111 1111 64 Mbytes 000 0000 1111 2 Mbytes 011 1111 1111 128 Mbytes 000 0001 1111 4 Mbytes 111 1111 1111 256 Mbytes
	30	Vs	Supervisor mode valid bit. Interacts with MSR[PR] to determine if there is a match with the logical address. See Section 7.5.2, “Recognition of Addresses in BAT Arrays.”
	31	Vp	User mode valid bit. Interacts with MSR[PR] to determine if there is a match with the logical address. See Section 7.5.2, “Recognition of Addresses in BAT Arrays.”
Lower BAT Register	0–14	BRPN	Physical (real) block number. Used with BL to generate high-order bits of the physical address of the block.
	15–24	—	Reserved
	25–28	WIMG	Memory/cache access mode bits W Write-through I Caching-inhibited M Memory coherence G Guarded Attempting to write to the W and G bits in IBATs causes boundedly-undefined results. For more information about the WIMG bits, see Section 5.3.1, “Memory/Cache Access Attributes.”
	29	—	Reserved
	30–31	PP	Protection bits. Determines protection for the block, as described in Section 7.5.4, “Block Memory Protection.”

7.5.4 Block Memory Protection

After an effective address is determined to be within a block defined by the BAT array, the access is validated by the memory protection mechanism. If this mechanism prohibits the access, a block protection violation exception (DSI or ISI) is generated.

The memory protection mechanism allows selectively granting read access, granting read/write access, and prohibiting access to areas of memory based on a number of control criteria. The block protection mechanism provides protection at the granularity defined by the block size (128 Kbytes to 256 Mbytes).

As the memory protection mechanism used by the block and page address translation is different, refer to Section 7.6.4, “Page Memory Protection,” for specific information unique to page address translation.

For block address translation, the memory protection mechanism is controlled by the PP bits located in the lower BAT register. The PP bits define access options for the block. Table 7-8 shows the types of accesses allowed for the possible PP bit combinations.

Table 7-8. Access Protection Control for Blocks

PP	Accesses Allowed
00	No access
x1	Read only
10	Read/write

Thus, any access attempted (read or write) when PP = 00 results in a protection violation exception condition. When PP = x1, an attempt to perform a write access causes a protection violation exception condition, and when PP = 10, all accesses are allowed. When the memory protection mechanism prohibits a reference, one of the following occurs, depending on the type of access that was attempted:

- For data accesses, a DSI is generated and DSISR[4] is set.
- For instruction accesses, an ISI is generated and SRR1[4] is set.

See Chapter 6, “Interrupts,” for more information.

Table 7-9 summarizes the conditions that cause interrupts for supervisor and user read and write accesses within a BAT area. Each BAT array entry is programmed to be either used or ignored for supervisor and user accesses via the BAT array entry valid bits, and the PP bits enforce the read/write protection options. Note that the valid bits (Vs and Vp) are used as part of the match criteria for a BAT array entry and are not explicitly part of the protection mechanism.

Table 7-9. Access Protection Summary for BAT Array

Vs	Vp	PP Field	Block Type	User Read	User Write	Supervisor Read	Supervisor Write
0	0	xx	No BAT array match	Not used	Not used	Not used	Not used
0	1	00	User—no access	Interrupt	Interrupt	Not used	Not used
0	1	x1	Use-–read-only	√	Interrupt	Not used	Not used
0	1	10	User—read/write	√	√	Not used	Not used
1	0	00	Supervisor—no access	Not used	Not used	Interrupt	Interrupt

Table 7-9. Access Protection Summary for BAT Array (continued)

Vs	Vp	PP Field	Block Type	User Read	User Write	Supervisor Read	Supervisor Write
1	0	x1	Supervisor—read-only	Not used	Not used	√	Interrupt
1	0	10	Supervisor—read/write	Not used	Not used	√	√
1	1	00	Both—no access	Interrupt	Interrupt	Interrupt	Interrupt
1	1	x1	Both—read-only	√	Interrupt	√	Interrupt
1	1	10	Both—read/write	√	√	√	√

Note: The term ‘not used’ implies that the access is not translated by the BAT array and is translated by the page address translation mechanism described in Section 7.6, “Memory Segment Model,” instead.

Note that because access to the BAT registers is privileged, only supervisor programs can modify the protection and valid bits for the block.

Figure 7-10 expands on the actions the processor takes in response to a memory protection violation. Note that **dcbt** and **dcbtst** do not cause interrupts; in the case of a memory protection violation for the attempted execution of one of these instructions, the translation is aborted and the instruction executes as a no-op (no violation is reported). Section 6.5.3, “Data Storage Interrupt (0x00300),” and Section 6.5.4, “Instruction Storage Interrupt (0x00400),” describes SRR1 and DSISR bit settings for protection violation exceptions.

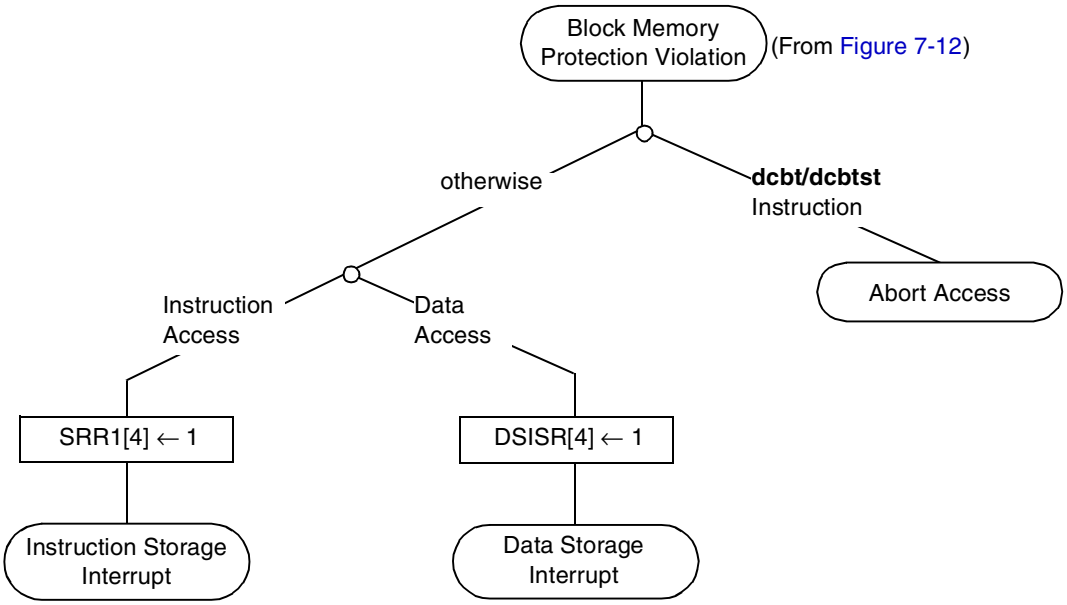


Figure 7-10. Memory Protection Violation Flow for Blocks

7.5.5 Block Physical Address Generation

Access to the physical memory within the block is made according to the memory/cache access mode defined by the WIMG bits in the lower BAT register. These bits apply to the entire block rather than to an individual page as described in Section 5.3.1, “Memory/Cache Access Attributes.”

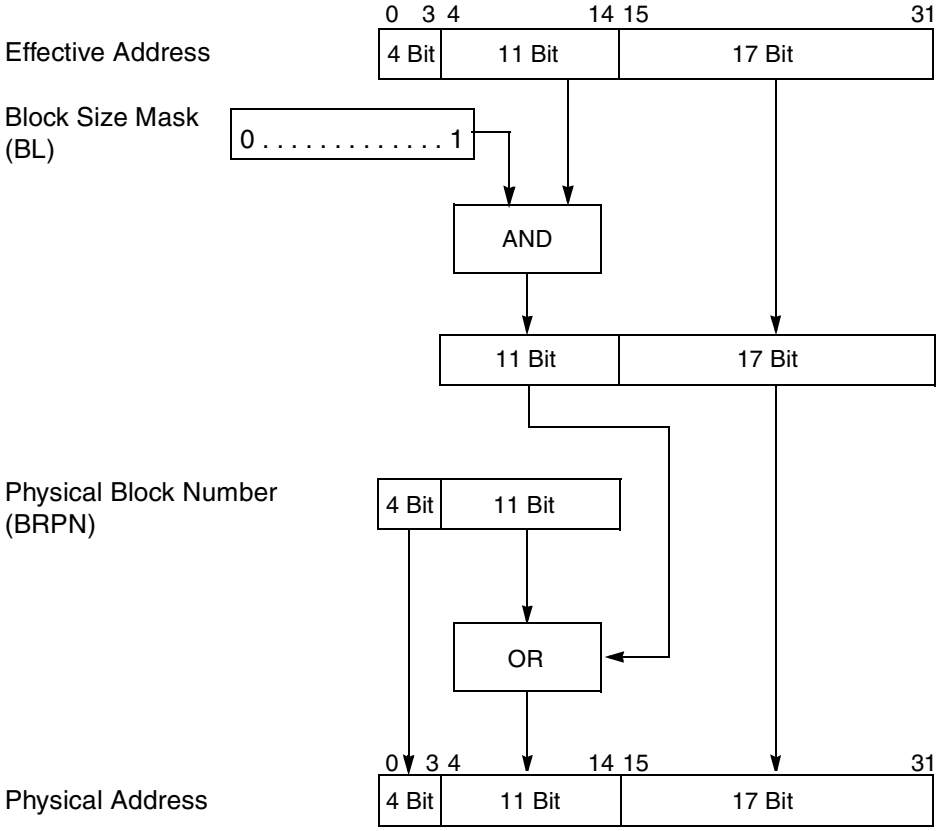


Figure 7-11. Block Physical Address Generation

7.5.6 Block Address Translation Summary

Figure 7-12 is an expansion of the BAT array hit branch of Figure 7-3 and shows the address bit translation. Note that the figure does not show when many of the interrupts in Figure 7-4 are detected or taken as this is implementation-specific.

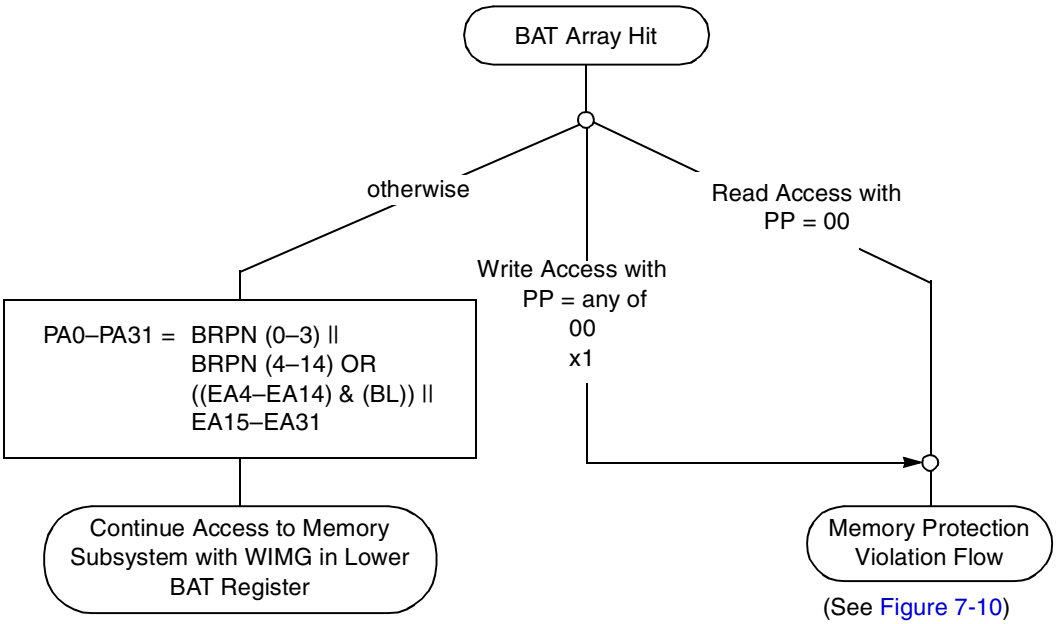


Figure 7-12. Block Address Translation Flow

7.6 Memory Segment Model

The OEA divides memory into 256-Mbyte segments. This segmented memory model provides a way to map 4-Kbyte pages of effective addresses to 4-Kbyte pages in physical memory (page address translation), while providing the programming flexibility afforded by a large virtual address space (52 bits).

A page address translation may be superseded by a matching block address translation as described in Section 7.5, “Block Address Translation.” If not, the page translation proceeds in the following two steps:

1. From effective address to the virtual address (which never exists as a specific entity but can be considered to be the concatenation of the virtual page number and the byte offset within a page)
2. From virtual address to physical address

The page address translation mechanism is described in the following sections, followed by a summary of page address translation with a detailed flow diagram.

7.6.1 Recognition of Addresses in Segments

The page address translation uses segment registers (SRs), which provide virtual address and protection information, and page table entries (PTEs), which provide the physical address and page protection information. The SRs are programmed by the operating system to provide the virtual ID for a segment. In

addition, the operating system also creates the page table in memory that provides the virtual-to-physical address mappings (in the form of PTEs) for the pages in memory.

Segments in the OEA can be classified as one of the following two types:

- Memory segment—An effective address in these segments represents a virtual address that is used to define the physical address of the page.
- Direct-store segment—References made to direct-store segments do not use the virtual paging mechanism of the processor. Note that the direct-store facility is optional and being removed from the architecture. See Section 7.8, “Direct-Store Segment Address Translation,” for a complete description of the mapping of direct-store segments for those processors that implement it.

SR[T] selects between memory segments and direct-store segments, as shown in Figure 7-10.

Table 7-10. Segment Register Types

SR[T]	Segment Type
0	Memory segment
1	Direct-store segment—optional, being removed from the architecture. Its use is discouraged.

7.6.1.1 Selection of Memory Segments

All accesses generated by the processor can be mapped to an SR; however, if translation is disabled (MSR[IR] = 0 or MSR[DR] = 0 for an instruction or data access, respectively), real addressing mode translation is performed as described in Section 7.4, “Real Addressing Mode.” Otherwise, if T = 0 in the corresponding SR (and the address is not translated by the BAT mechanism), the access maps to memory space and page address translation is performed.

After a memory segment is selected, the processor creates the virtual address for the segment and searches for the PTE that dictates the physical page number to be used for the access. Note that I/O devices can be easily mapped into memory space and used as memory-mapped I/O.

7.6.1.2 Selection of Direct-Store Segments

As described for memory segments, all accesses generated by the processor (with translation enabled) map to an SR. If T = 1 for the selected SR, the access maps to the direct-store interface space and the access proceeds as described in Section 7.8, “Direct-Store Segment Address Translation.” Because the direct-store interface is present only for compatibility with existing I/O devices that used this interface and because the direct-store interface protocol is not optimized for performance, its use is discouraged. Additionally, future devices are not likely to support it. Thus, software should not depend on its results and new software should not use it. The most efficient method for accessing I/O is by mapping the I/O areas to memory segments.

7.6.2 Page Address Translation Overview

The translation of effective addresses to physical addresses is shown in Figure 7-13. The address translation is as follows:

- EA[0–3] comprise the segment register number used to select an SR, from which the virtual segment ID (VSID) is extracted.
- EA[4–19] correspond to the page number within the segment; these are concatenated with SR[VSID] to form the virtual page number (VPN), which is used to search for the PTE in either an on-chip TLB or the page table. The PTE then provides the physical page number (RPN).
- EA[20–31] are the byte offset within the page; these are concatenated with the RPN field of a PTE to form the physical address used to access memory.

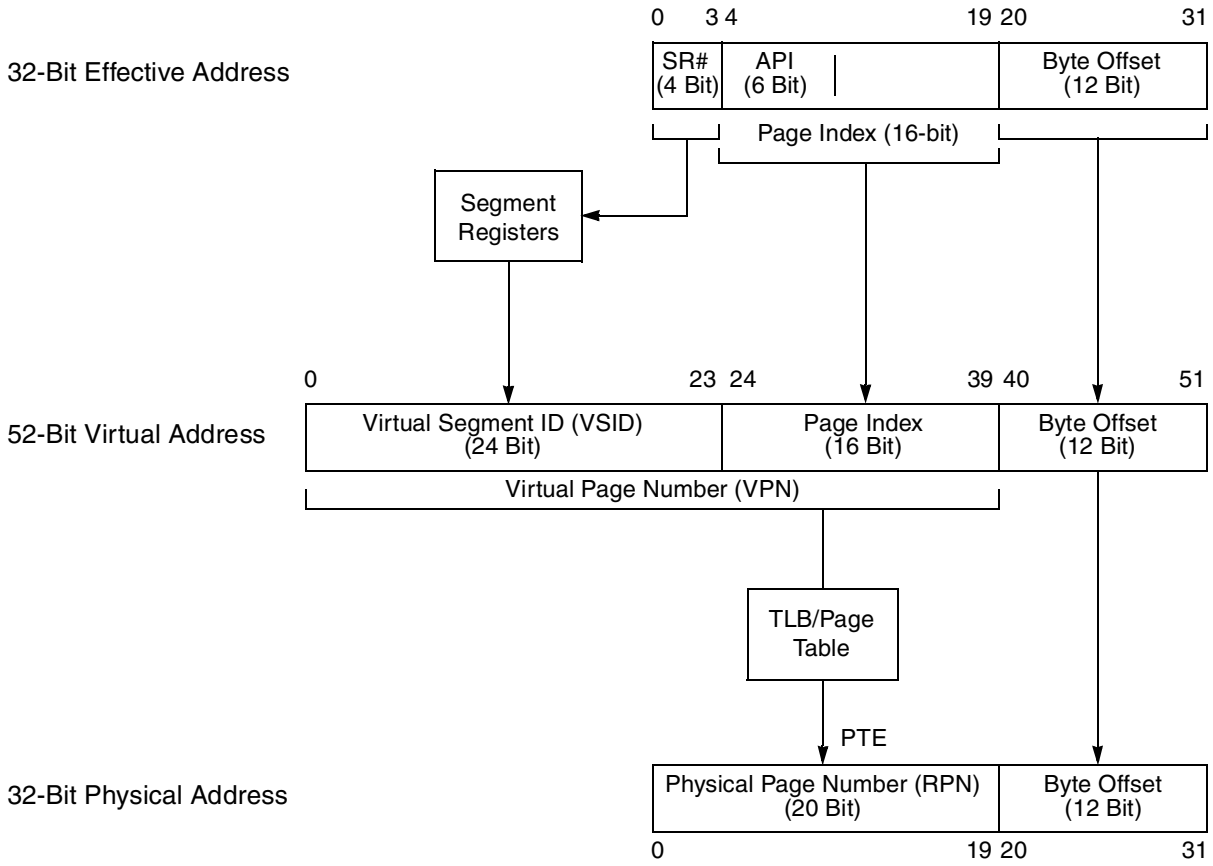


Figure 7-13. Page Address Translation Overview

7.6.2.1 Segment Register Definitions

The SR fields are interpreted differently depending on the value of the T bit within the descriptor. When T = 1, the SR defines a direct-store segment and the format is as described in Section 7.8.1, “Segment Registers for Direct-Store Segments.”

7.6.2.1.1 Segment Register Format

Figure 7-14 shows the segment register format used in page address translation (T = 0).

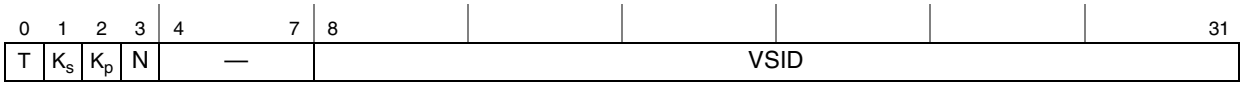


Figure 7-14. Segment Register Format for Page Address Translation

Table 7-11 provides the corresponding bit definitions of the segment register.

Table 7-11. Segment Register Bit Definition for Page Address Translation

Bit	Name	Description
0	T	T = 0 selects this format
1	Ks	Supervisor-state protection key
2	Kp	User-state protection key
3	N	No-execute protection bit
4–7	—	Reserved
8–31	VSID	Virtual segment ID

The Ks and Kp bits partially define the access protection for the pages within the segment. The page protection provided in the OEA is described in Section 7.6.4, “Page Memory Protection.” The virtual segment ID field is used as the high-order bits of the virtual page number (VPN) as shown in Figure 7-13.

Instructions for accessing the SRs are summarized in Table 7-5. These instructions are executable only in supervisor mode. See Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for information about the synchronization requirements when modifying the SRs. See Chapter 8, “Instruction Set,” for more detail on the encodings of these instructions.

Note that the SRs described here are merely a conceptual model; a processor may implement separate SRs for instructions and for data, for example. In this case, it is the responsibility of the hardware to maintain the consistency between the multiple sets of SRs.

7.6.2.2 Page Table Entry (PTE) Definitions

PTEs are generated and placed in page table in memory by the operating system using the hashing algorithm described in Section 7.7.1.3, “Page Table Hashing Functions.” The OEA defines PTEs that are 64 bits in length. Some of the fields are defined as follows:

- The virtual segment ID field corresponds to the high-order VPN bits, and along with the H, V, and API fields, it is used to locate the PTE (used as match criteria in comparing the PTE with the segment information).
- The R and C bits maintain history information for the page as described in Section 7.6.3, “Page History Recording.”
- The WIMG bits define the memory/cache control mode for accesses to the page.
- The PP bits define the remaining access protection constraints for the page. Page protection is described in Section 7.6.4, “Page Memory Protection.”

Conceptually, the page table in memory must be searched to translate the address of every reference. For performance reasons, however, some processors use on-chip TLBs to cache copies of recently-used PTEs, eliminating table search time for most accesses. In this case, first the TLB is searched for the address translation. If a copy of the PTE is found, no page table search is performed. As TLBs are noncoherent caches of PTEs, software that changes the page table in any way must perform the appropriate TLB invalidate operations to keep the on-chip TLBs coherent with respect to the page table in memory.

7.6.2.2.1 PTE Format

Figure 7-15 shows the format of the two words that comprise a PTE.

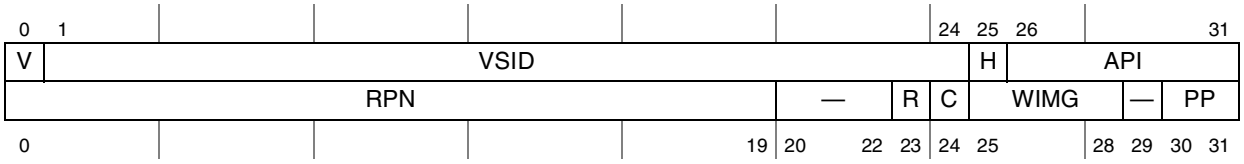


Figure 7-15. Page Table Entry Format

Table 7-12 lists the corresponding bit definitions for each word in a PTE as defined above.

Table 7-12. PTE Bit Definitions

Word	Bit	Name	Description
0	0	V	Entry valid (V = 1) or invalid (V = 0)
	1–24	VSID	Virtual segment ID
	25	H	Hash function identifier
	26–31	API	Abbreviated page index
1	0–19	RPN	Physical page number
	20–22	—	Reserved
	23	R	Reference bit
	24	C	Change bit
	25–28	WIMG	Memory/cache control bits
	29	—	Reserved
	30–31	PP	Page protection bits

In this case, the PTE contains an abbreviated page index rather than the complete page index field because at least ten of the low-order bits of the page index are used in the hash function to select a PTEG address (PTEG addresses define the location of a PTE). Therefore, these 10 low-order bits are not repeated in the PTEs of that PTEG.

7.6.3 Page History Recording

Reference (R) and change (C) bits in each PTE keep history information about the page. The operating system then uses this information to determine which areas of memory to write back to disk when new pages must be allocated in main memory. Reference and change recording is performed only for accesses

made with page address translation and not for translations made with the BAT mechanism or for accesses that correspond to direct-store (T = 1) segments. Furthermore, R and C bits are maintained only for accesses made while address translation is enabled (MSR[IR] = 1 or MSR[DR] = 1).

In general, the reference and change bits are updated to reflect the status of the page based on the access, as shown in Table 7-13.

Table 7-13. Table Search Operations to Update History Bits

R and C Bits	Processor Action
00	Read: Table search operation to update R Write: Table search operation to update R and C
01	Combination does not occur
10	Read: No special action Write: Table search operation to update C
11	No special action for read or write

Processors that implement a TLB may perform the R and C bit updates based on the copies of these bits resident in the TLB. For example, the processor may update the C bit based only on the status of the C bit in the TLB entry in the case of a TLB hit (the R bit may be assumed to be set in the page tables if there is a TLB hit). Therefore, when software clears the R and C bits in the page tables in memory, it must invalidate the TLB entries associated with the pages whose reference and change bits were cleared. See Section 7.7.3, “Page Table Updates,” for all of the constraints imposed on the software when updating the reference and change bits in the page tables.

The R bit for a page may be set by the execution of the **dcbt** or **dcbtst** instruction to that page. However, neither of these instructions cause the C bit to be set.

The update of the reference and change bits is performed as if address translation were disabled (real addressing mode address).

7.6.3.1 Reference Bit

The reference bit for each virtual page is located in the PTE. Every time a page is reference (by an instruction fetch or any read or write access) the reference bit is set in the page table. The reference bit may be set immediately, or the setting may be delayed until the memory access is determined to be successful. Because the reference to a page is what causes a PTE to be loaded into the TLB, some processors may assume the R bit in the TLB is always set. The processor never automatically clears the reference bit.

The reference bit is only a hint to the operating system about the activity of a page. At times, the reference bit may be set although the access was not logically required by the program or even if the access was prevented by memory protection. Examples of this include the following:

- Fetching of instructions not subsequently executed
- Accesses generated by an **lswx** or **stswx** instruction with a zero length
- Accesses generated by an **stwcx.** instruction when no store is performed
- Accesses that cause interrupts and are not completed

7.6.3.2 Change Bit

The change bit for each virtual page is located both in the PTE in the page table and in the copy of the PTE loaded into the TLB (if a TLB is implemented). Whenever a data store instruction is executed successfully, if the TLB search (for page address translation) results in a hit, the change bit in the matching TLB entry is checked. If it is already set, it is not updated. If the TLB change bit is 0, it is set and a table search operation is performed to set the C bit in the corresponding PTE in the page table.

Processors cause the change bit (in both the PTE in the page tables and in the TLB if implemented) to be set only when the page memory protection mechanism allows a store operation that is guaranteed to be in the execution path, unless an interrupt, other than those caused by one of the following occurs:

- System-caused interrupts (system reset, machine check, external, and decrementer interrupts)
- Floating-point enabled interrupt type program interrupts when the processor is in an imprecise mode
- Floating-point assist interrupts for instructions that cause no other kind of precise interrupt

Furthermore, the following conditions may cause the C bit to be set:

- The execution of an **stwcx.** instruction is allowed by the memory protection mechanism but a store operation is not performed.
- The execution of a **stswx** instruction is allowed by the memory protection mechanism, but a store operation is not allowed because the specified length is zero.
- A **dcba** or **dcbi** instruction is executed.

No other cases cause the C bit to be set.

7.6.3.3 Scenarios for Reference and Change Bit Recording

This section summarizes the OEA-defined model that automatically maintains reference and change bits in hardware, in the setting of the R and C bits. In some scenarios, the bits are guaranteed to be set by the processor; in some scenarios, the architecture allows that the bits may be set (not absolutely required); and in some scenarios, the bits are guaranteed to not be set. Note that when the hardware updates the R and C bits in memory, the accesses are performed as a physical memory access, as if the WIMG bit settings were 0b0010 (that is, as unguarded cacheable operations in which coherency is required).

In implementations that do not maintain the R and C bits in hardware, software assistance is required. For these processors, the information in this section still applies, except that the software performing the updates is constrained to the rules described (that is, must set bits shown as guaranteed to be set and must not set bits shown as guaranteed to not be set). Note that this software should be contained in the area of memory reserved for implementation-specific use and should be invisible to the operating system.

Table 7-14 defines a prioritized list of the R and C bit settings for all scenarios. The entries in the table are prioritized from top to bottom, such that a matching scenario occurring closer to the top of the table takes precedence over one closer to the bottom. For example, if a **stwcx.** instruction causes a protection violation and there is no reservation, the C bit is not altered, as shown for the protection violation case. Note that in the table, load operations include those generated by load instructions, by the **eciwx** instruction, and by the cache management instructions that are treated as loads. Similarly, store operations include those operations generated by store instructions, by the **ecowx** instruction, and by these cache management instructions that are treated as stores.

Table 7-14. Model for Guaranteed R and C Bit Settings

Priority	Scenario	Causes Setting of R Bit	Causes Setting of C Bit
1	No-execute protection violation	No	No
2	Page protection violation	Maybe	No
3	Speculative instruction fetch or load operation	Maybe	No
4	Speculative [1] store operation for instructions that cause no other kind of precise interrupt (in the absence of system-caused, imprecise, or floating-point assist interrupts)	Maybe [2]	Maybe [2]
5	All other speculative [1] store operations	Maybe[2]	No
6	Zero-length load (**lswx**)	Maybe	No
7	Zero-length store (**stswx**)	Maybe [2]	Maybe [2]
8	Store conditional (**stwcx.**) that does not store	Maybe [2]	Maybe [2]
9	In-order instruction fetch	Yes [3]	No
10	Load instruction or **eciwx**	Yes	No
11	Store instruction, **ecowx**, **dcbz**, or **dcba** [4] instruction	Yes	Yes
12	**icbi**, **dcbt**, **dcbtst**, **dcbst**, or **dcbf** instruction	Maybe	No
13	**dcbi** instruction	Maybe [2]	Maybe [2]

1 Referred to as out-of-order in the architecture.

2 If C is set, R is guaranteed to also be set.

3 This includes the case in which the instruction was fetched speculatively and R was not set.

4 For a **dcba** instruction that does not modify the target block, it is possible that neither bit is set.

7.6.3.4 Synchronization of Memory Accesses and Reference and Change Bit Updates

Although the processor updates the reference and change bits in the page tables automatically, these updates are not guaranteed to be immediately visible to the program after the load, store, or instruction fetch operation that caused the update. If processor A executes a load or store or fetches an instruction, the following conditions are met with respect to performing the access and updating R and C bit:

- If processor A subsequently executes a **sync** instruction, both the updates to the bits in the page table and the load or store operation are guaranteed to be performed with respect to all processors and mechanisms before the **sync** instruction completes on processor A.
- Additionally, if processor B executes a **tlbie** instruction that does the following:
  - — Signals the invalidation to the hardware
  - — Invalidates the TLB entry for the access in processor A
  - — Is detected by processor A after processor A has begun the access

  and processor B executes a **tlbsync** instruction after it executes the **tlbie** both the updates to the bits and the original access are guaranteed to be performed with respect to all processors and mechanisms before the **tlbsync** instruction completes on processor A.

7.6.4 Page Memory Protection

In addition to the no-execute option that can be programmed at the SR level to prevent instructions from being fetched from a given segment (shown in Figure 7-4), there are a number of other memory protection options that can be programmed at the page level. The page memory protection mechanism allows selectively granting read access, granting read/write access, and prohibiting access to areas of memory based on a number of control criteria.

The memory protection used by the block and page address translation mechanisms is different in that the page address translation protection defines a key bit that, in conjunction with the PP bits, determines whether supervisor and user programs can access a page. For specific information about block address translation, refer to Section 7.5.4, “Block Memory Protection.”

For page address translation, the memory protection mechanism is controlled by the following:

- MSR[PR]. MSR[PR] = 0 selects supervisor mode; MSR[PR] = 1 selects user mode.
- SR*n*[Ks] and SR*n*[Kp], the supervisor and user key bits, which define the key for the page
- The PP bits, located in the PTE, define the access options for the page

The key bits, the PP bits, and the MSR[PR] bit are used as follows:

- When an access is generated, one of the key bits is selected to be the key as follows:
  - For supervisor accesses (MSR[PR] = 0), the Ks bit is used and Kp is ignored
  - For user accesses (MSR[PR] = 1), the Kp bit is used and Ks is ignored

  That is, key = (Kp & MSR[PR]) | (Ks & ¬MSR[PR])
- The selected key is used with the PP bits to determine if instruction fetching, load access, or store access is allowed.

Table 7-15 shows the types of accesses that are allowed for the general case (all possible Ks, Kp, and PP bit combinations), assuming that SR[N] is cleared (the no-execute option is not selected).

Table 7-15. Access Protection Control with Key

Key [1]	PP[2]	Page Type
0	00	Read/write
0	01	Read/write
0	10	Read/write
0	11	Read only
1	00	No access
1	01	Read only
1	10	Read/write
1	11	Read only

1 Ks or Kp selected by state of MSR[PR]

2 PP protection option bits in PTE

Thus, the conditions that cause a protection violation (not including the no-execute protection option for instruction fetches) are depicted in Table 7-16 and as a flow diagram in Figure 7-18. Any access attempted (read or write) when the key = 1 and PP = 00, causes a protection violation exception. When key = 1 and PP = 01, an attempt to perform a write access causes a protection violation exception. When PP = 10, all accesses are allowed, and when PP = 11, write accesses always cause an interrupt. The processor takes either the ISI or the DSI (for an instruction or data access, respectively) when there is an attempt to violate the memory protection.

Table 7-16. Exceptions for Key and PP Combinations

Key	PP	Prohibited Accesses
0	0x	None
1	00	Read/write
1	01	Write
x	10	None
x	11	Write

Any combination of the Ks, Kp, and PP bits is allowed. One example is if the Ks and Kp bits are programmed so that the value of the key bit for Table 7-15 directly matches the MSR[PR] bit for the access. In this case, the encoding of Ks = 0 and Kp = 1 is used for the PTE, and the PP bits then enforce the protection options shown in Table 7-17.

Table 7-17. Access Protection Encoding of PP Bits for Ks = 0 and Kp = 1

PP Field	Option	Key = 1		Key = 0	
		User Read	User Write	Supervisor Read	Supervisor Write
00	Supervisor-only	Violation	Violation	√	√
01	Supervisor write-only	√	Violation	√	√
10	Both user/supervisor	√	√	√	√
11	Both read-only	√	Violation	√	Violation

However, if the setting Ks = 1 is used, supervisor accesses are treated as user reads and writes with respect to Table 7-17. Likewise, if the setting Kp = 0 is used, user accesses to the page are treated as supervisor accesses in relation to Table 7-17. Therefore, by modifying one of the key bits (in the SR), the way the processor interprets accesses (supervisor or user) in a particular segment can easily be changed. Note, however, that only supervisor programs are allowed to modify the SR key bits.

When the memory protection mechanism prohibits a reference, the flow of events is similar to that for a memory protection violation occurring with the block protection mechanism. As shown in Figure 7-16, one of the following occurs depending on the type of access that was attempted:

- For data accesses, a DSI interrupt is generated and DSISR[4] is set. If the access is a store, DSISR[6] is also set.
- For instruction accesses, the following occurs:
  - An ISI is generated and SRR1[4] is set.
  - If the segment is designated as no-execute, an ISI is generated and SRR1[3] is set.

The only difference between the flow shown in Figure 7-16 and that of the block memory protection violation is the ISI that can be caused by an attempt to fetch an instruction from a segment that has been designated as no-execute (SR[N] = 1). See Section 6.5.3, “Data Storage Interrupt (0x00300),” and Section 6.5.3, “Data Storage Interrupt (0x00300),” for more information.

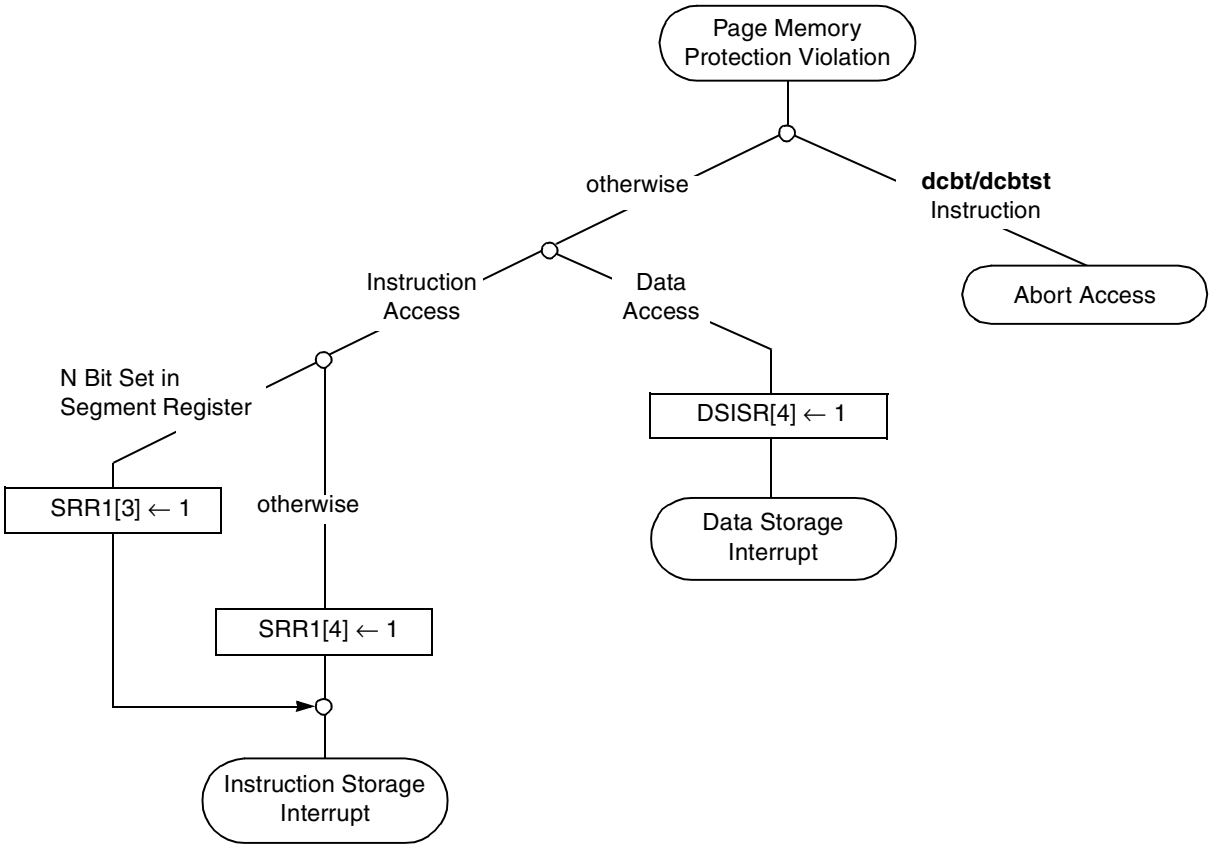
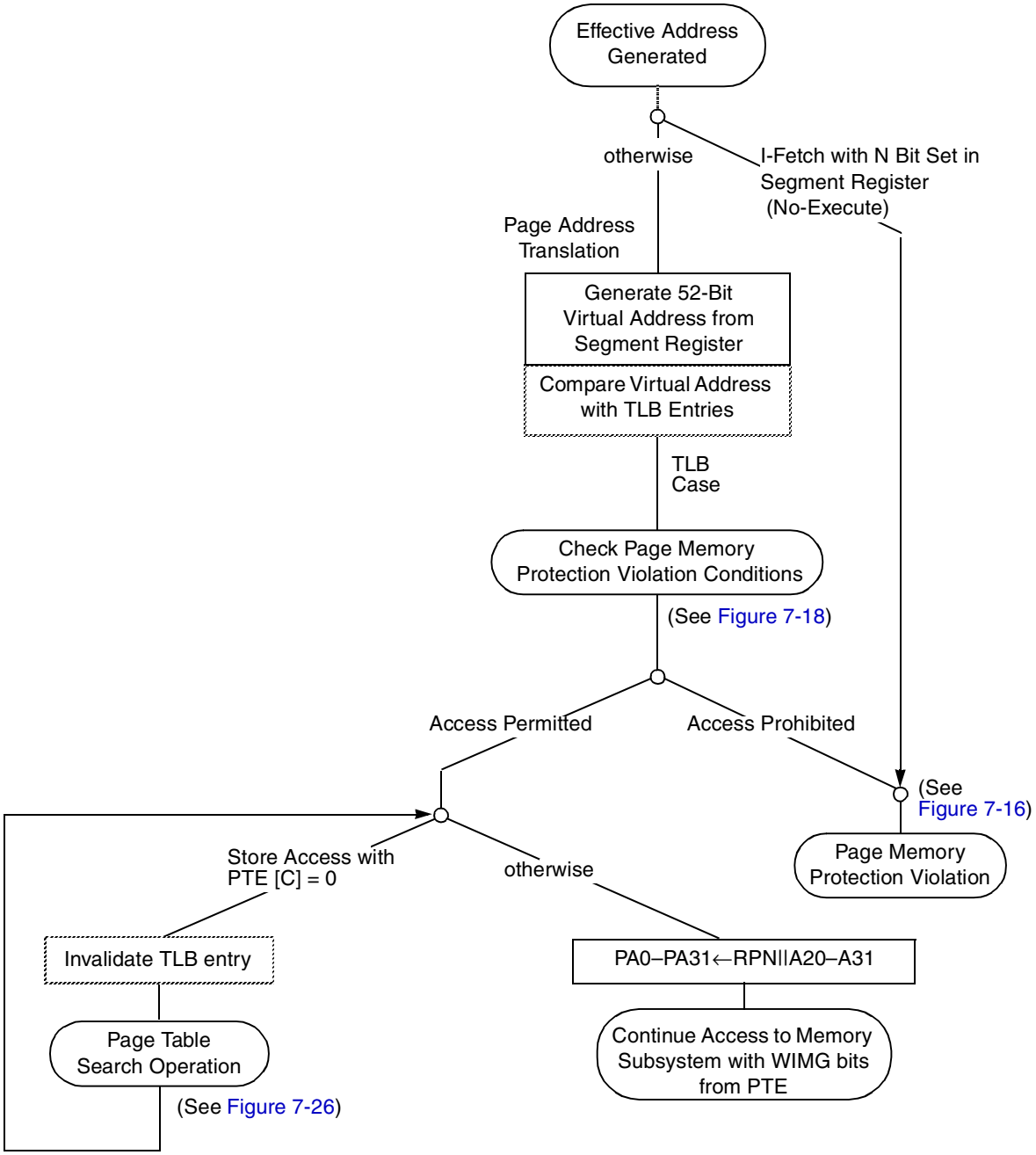


Figure 7-16. Memory Protection Violation Flow for Pages

If the page protection mechanism prohibits a store operation, the C bit is not set (in either the TLB or in the page tables in memory); however, a prohibited store access may cause a PTE to be loaded into the TLB and consequently cause the R bit to be set in a PTE (both in the TLB and in the page table in memory).

7.6.5 Page Address Translation Summary

Figure 7-17 provides the detailed flow for the page address translation mechanism. The figure includes the checking of SR*n*[N] and then expands on the ‘TLB Hit’ branch of Figure 7-4. The detailed flow for the ‘TLB Miss’ branch of Figure 7-4 is described in Section 7.7.2, “Page Table Search Operation.” The checking of memory protection violation conditions for page address translation is shown in Figure 7-18. The ‘Invalidate TLB Entry’ box shown in Figure 7-17 is marked as implementation-specific as this level of detail for TLBs (and the existence of TLBs) is not dictated by the architecture. Note that the figure does not show the detection of all exceptions shown in Table 7-3 and Figure 7-4; the flow for many of these interrupts is implementation-specific.



Note: ..... Implementation-specific

Figure 7-17. Page Address Translation Flow—TLB Hit

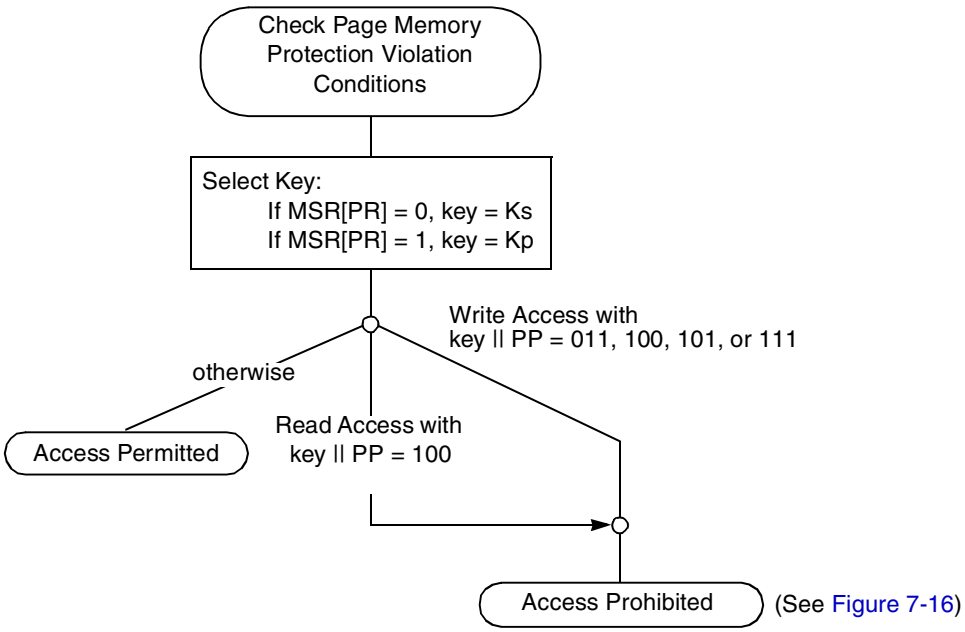


Figure 7-18. Page Memory Protection Violation Conditions for Page Address Translation

7.7 Hashed Page Tables

If a copy of the PTE corresponding to the VPN for an access is not resident in a TLB (corresponding to a miss in the TLB, provided a TLB is implemented), the processor must search for the PTE in the page tables set up by the operating system in main memory.

The algorithm specified by the architecture for accessing the page tables includes a hashing function on some of the virtual address bits. Thus, the addresses for PTEs are allocated more evenly within the page tables and the hit rate of the page tables is maximized. This algorithm must be synthesized by the operating system for it to correctly place the page table entries in main memory.

If page table search operations are performed automatically by the hardware, they are performed using physical addresses and as if the memory access attribute bit M = 1 (memory coherency enforced in hardware). If the software performs the page table search operations, the accesses must be performed in real addressing mode (MSR[DR] = 0); this additionally guarantees that M = 1.

This section describes the format of the page tables and the algorithm used to access them. In addition, the constraints imposed on the software in updating the page tables (and other MMU resources) are described.

7.7.1 Page Table Definition

The hashed page table is a variable-sized data structure that defines the mapping between virtual page numbers and physical page numbers. The page table size is a power of 2, its starting address is a multiple of its size, and the table must reside in memory with the WIMG attributes of 0b0010.

The page table contains a number of page table entry groups (PTEGs). A PTEG contains eight PTEs of 8 bytes each; therefore, each PTEG is 64 bytes long. PTEG addresses are entry points for table search

operations. Figure 7-19 shows two PTEG addresses (PTEGaddr1 and PTEGaddr2) where a given PTE may reside.

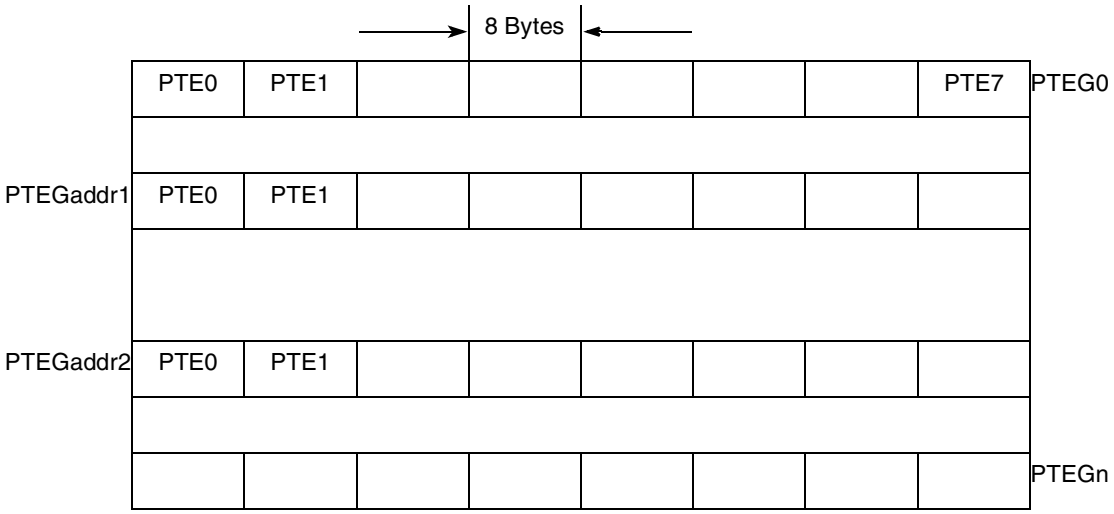


Figure 7-19. Page Table Definitions

A given PTE can reside in one of two PTEGS—the primary PTEG or the secondary PTEG. Additionally, a given PTE can reside in any of the PTE locations within an addressed PTEG. Thus, a given PTE may reside in one of 16 possible locations within the page table. If a given PTE is neither the primary or secondary PTEG, a page table miss occurs, corresponding to a page fault condition.

A table search operation is defined as the search for a PTE within a primary and secondary PTEG. When a table search operation commences, a primary hashing function is performed on the virtual address. The output of the hashing function is then concatenated with bits programmed into SDR1 by the operating system to create the physical address of the primary PTEG. The PTEs in the PTEG are then checked, one by one, to see if there is a hit within the PTEG. If the PTE is not located, a secondary hashing function is performed, a new physical address is generated for the PTEG, and the PTE is searched for again, using the secondary PTEG address.

Note, however, that although a given PTE may reside in one of 16 possible locations, an address that is a primary PTEG address for some accesses also functions as a secondary PTEG address for a second set of accesses (as defined by the secondary hashing function). Therefore, these 16 possible locations are really shared by two different sets of effective addresses. Section 7.7.1.6, “Page Table Structure Examples,” illustrates how PTEs map into the 16 possible locations as primary and secondary PTEs.

7.7.1.1 SDR1 Register Definitions

SDR1 contains the control information for the page table structure in that it defines the high-order bits for the physical base address of the page table and it defines the size of the table. Note that there are certain synchronization requirements for writing to SDR1 that are described in Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers.”

Figure 7-20 shows the format of the SDR1 register.

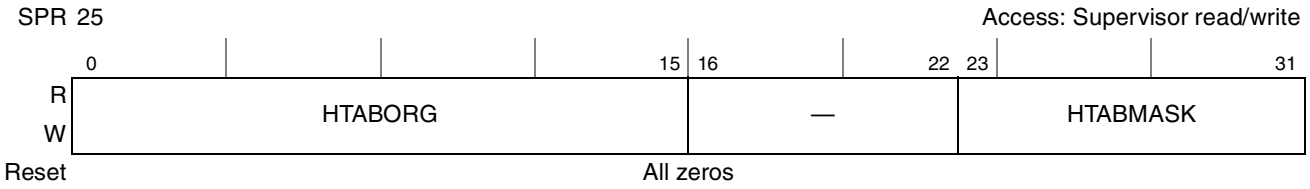


Figure 7-20. SDR1 Register Format

Bit settings are described in Figure 7-18.

Table 7-18. SDR1 Register Bit Settings

Bits	Name	Description
0–15	HTABORG	Physical base address of page table
16–22	—	Reserved
23–31	HTABMASK	Mask for page table address

SDR1[HTABORG] contains the high-order 16 bits of the 32-bit physical address of the page table. Therefore, the beginning of the page table lies on a 2^{16} byte (64 Kbyte) boundary at a minimum. If the processor does not support 32 bits of physical address, software should write zeros to those unsupported bits in the HTABORG field (as the implementation treats them as reserved). Otherwise, a machine check interrupt can occur.

A page table can be any size 2^n bytes where $16 \le n \le 25$. SDR1[HTABMASK] contains a mask value that determines how many bits from the output of the hashing function are used as the page table index. This mask must be of the form 0b00...011...1 (a string of 0 bits followed by a string of 1 bits). As the table size increases, more bits are used from the output of the hashing function to index into the table. The 1 bits in HTABMASK determine how many additional bits (beyond the minimum of 10) from the hash are used in the index; the HTABORG field must have the same number of low-order bits equal to 0 as the HTABMASK field has low-order bits equal to 1.

For example, suppose that the page table is 16,384 (2^{14}) 64-byte PTEGs, for a total size of 2^{20} bytes (1 Mbyte). A 14-bit index is required. Ten bits are provided from the hash to start with, so 4 additional bits from the hash must be selected. Thus the value in HTABMASK must be 15 and the value in HTABORG must have its low-order 4 bits (SDR1[12–15]) equal to 0. This means that the page table must begin on a $2^{<4 + 10 + 6>} = 2^{20}$ = 1-Mbyte boundary.

7.7.1.2 Page Table Size

The number of entries in the page table directly affects performance because it influences the hit ratio in the page table and thus the rate of page fault exceptions. If the table is too small, not all virtual pages that have physical page frames assigned may be mapped via the page table. This can happen if more than 16 entries map to the same primary/secondary pair of PTEGs; in this case, many hash collisions may occur.

The minimum page table size is 64 Kbytes (2^{10} PTEGs of 64 bytes each). However, it is recommended that the total number of PTEGs in the page table be at least half the number of physical page frames to be mapped. While avoidance of hash collisions cannot be guaranteed for any size page table, making the page

table larger than the recommended minimum size reduces the frequency of such collisions by making the primary PTEGs more sparsely populated, and further reducing the need to use the secondary PTEGs.

Table 7-19 shows some example sizes for total main memory in a 32-bit system. The recommended minimum page table size for these example memory sizes are then outlined, along with their corresponding SDR1[HTABORG,HTABMASK] settings. Note that systems with less than 8 Mbytes of main memory may be designed with 32-bit processors, but the minimum amount of memory that can be used for the page tables in these cases is 64 Kbytes.

Table 7-19. Minimum Recommended Page Table Sizes

Total Main Memory	Recommended Minimum			Settings for Recommended Minimum	
	Memory for Page Tables	Number of Mapped Pages (PTEs)	Number of PTEGs	HTABORG (Maskable Bits 7–15)	HTABMASK
8 Mbytes (2^{23})	64 Kbytes (2^{16})	2^{13}	2^{10}	x xxxx xxxx	0 0000 0000
16 Mbytes (2^{24})	128 Kbytes (2^{17})	2^{14}	2^{11}	x xxxx xxx0	0 0000 0001
32 Mbytes (2^{25})	256 Kbytes (2^{18})	2^{15}	2^{12}	x xxxx xx00	0 0000 0011
64 Mbytes (2^{26})	512 Kbytes (2^{19})	2^{16}	2^{13}	x xxxx x000	0 0000 0111
128 Mbytes (2^{27})	1 Mbyte (2^{20})	2^{17}	2^{14}	x xxxx 0000	0 0000 1111
256 Mbytes (2^{28})	2 Mbytes (2^{21})	2^{18}	2^{15}	x xxx0 0000	0 0001 1111
512 Mbytes (2^{29})	4 Mbytes (2^{22})	2^{19}	2^{16}	x xx00 0000	0 0011 1111
1 Gbytes (2^{30})	8 Mbytes (2^{23})	2^{20}	2^{17}	x x000 0000	0 0111 1111
2 Gbytes (2^{31})	16 Mbytes (2^{24})	2^{21}	2^{18}	x 0000 0000	0 1111 1111
4 Gbytes (2^{32})	32 Mbytes (2^{25})	2^{22}	2^{19}	0 0000 0000	1 1111 1111

For example, if the physical memory size is 2^{29} bytes (512 Mbytes), there are $2^{29} – 2^{12}$ (4-Kbyte page size) = 2^{17} (128 Kbytes) total page frames. If this number of page frames is divided by 2, the resultant minimum recommended page table size is 2^{16} PTEGs, or 2^{22} bytes (4 Mbytes) of memory for the page tables.

7.7.1.3 Page Table Hashing Functions

The MMU uses a primary and a secondary hashing function in the creation of the physical addresses used in a page table search operation. These hashing functions distribute the PTEs within the page table, in that there are two possible PTEGs where a given PTE can reside. Additionally, a given PTE can reside at one of eight possible PTE locations within a PTEG. If a PTE is not found using the primary hashing function, the secondary hashing function is performed and the secondary PTEG is searched. Note that the operating system uses these two functions to set up the page tables in memory appropriately.

Typically, the hashing functions provide a high probability that a required PTE is resident in the page table without requiring the definition of all possible PTEs in main memory. However, if a PTE is not found in the secondary PTEG, a page fault occurs and an interrupt is taken. Thus, the required PTE can then be placed into either the primary or secondary PTEG by the system software, and on the next TLB miss to this page (in those processors that implement a TLB), the PTE will be found in the page tables (and loaded into an on-chip TLB).

The address of a PTEG is derived from SDR1[HTABORG], and the output of the corresponding hashing function (primary hashing function for primary PTEG and secondary hashing function for a secondary PTEG). The value in the HTABMASK field determines how many of the high-order hash value bits are masked and how many are used in the generation of the physical address of the PTEG.

Figure 7-21 shows the OEA-defined hashing functions. The inputs to the primary hashing function are the low-order 19 bits of the selected SR*n*[VSID] (bits 5–23 of the 52-bit virtual address), and the page index field of the effective address (bits 24–39 of the virtual address) concatenated with three zero high-order bits. The XOR of these two values generates the output of the primary hashing function (hash value 1).

When the secondary hashing function is required, the output of the primary hashing function is complemented with one's complement arithmetic, to provide hash value 2.

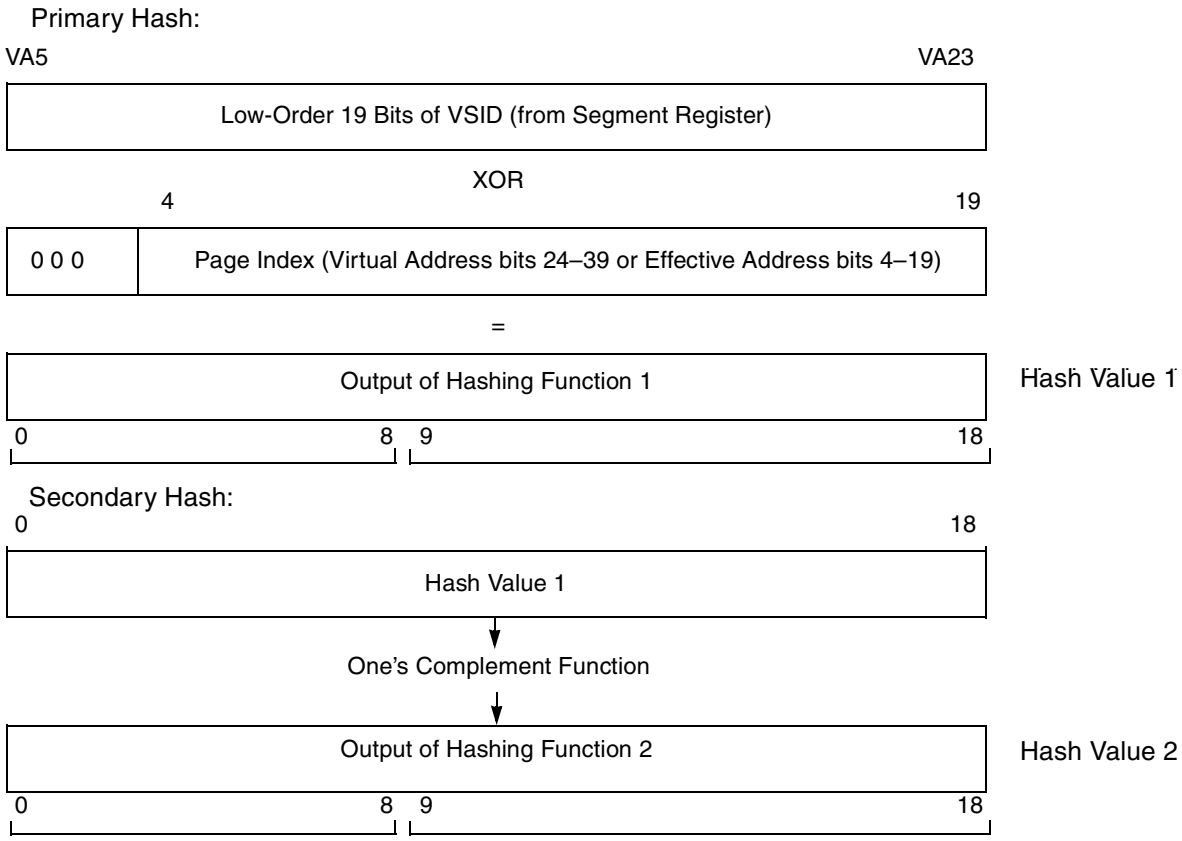


Figure 7-21. Hashing Functions for Page Tables

7.7.1.4 Page Table Addresses

The following sections illustrate the generation of the addresses used for accessing the hashed page tables. As stated earlier, the operating system must synthesize the table search algorithm for setting up the tables.

Two of the elements that define the virtual address (SR[VSID] and the page index field of the effective address) are used as inputs into a hashing function. Depending on whether the primary or secondary PTEG is to be accessed, the processor uses either the primary or secondary hashing function as described in Section 7.7.1.3, "Page Table Hashing Functions."

Note that unless all accesses to be performed by the processor can be translated by the BAT mechanism when address translation is enabled (MSR[DR] or MSR[IR] = 1), the SDR1 must point to a valid page table. Otherwise, a machine check interrupt can occur.

Additionally, care should be given that page table addresses not conflict with those that correspond to areas of the physical address map reserved for the interrupt vector table or other implementation-specific purposes (refer to Section 7.3.1.1, “Predefined Physical Memory Locations”).

The base address of the page table is defined by the high-order bits of SDR1[HTABORG].

Effectively, bits 7–15 of the PTEG address are derived from the masking of the high-order bits of the hash value (as defined by SDR1[HTABMASK]) concatenated with (implemented as an OR function) the high-order bits of SDR1[HTABORG] as defined by HTABMASK. Bits 16–25 of the PTEG address are the 10 low-order bits of the hash value, and bits 26–31 of the PTEG address are zero. In the process of searching for a PTE, the processor checks up to eight PTEs located in the primary PTEG and up to eight PTEs located in the secondary PTEG, if required, searching for a match. Figure 7-22 provides a graphical description of the generation of the PTEG addresses.

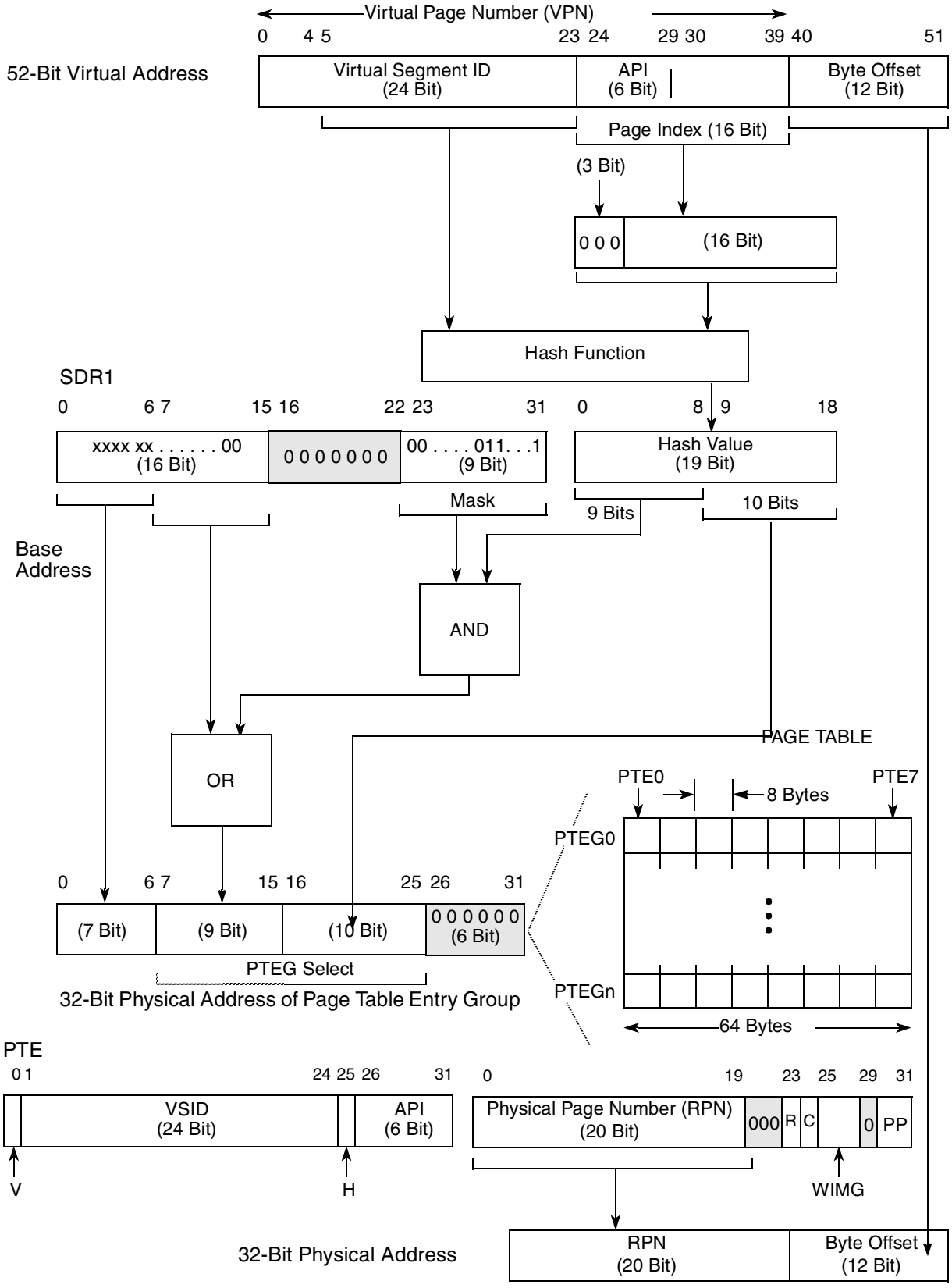


Figure 7-22. Generation of Addresses for Page Tables

7.7.1.5 Page Table Structure Summary

In the process of searching for a PTE, the processor interprets the values read from memory as described in Section 7.6.2.2, “Page Table Entry (PTE) Definitions.” The VSID and the abbreviated page index (API) fields of the virtual address of the access are compared to those same fields of the PTEs in memory. In addition, the valid (V) bit and the hashing function (H) bit are also checked. For a hit to occur, the V bit of the PTE in memory must be set. If the fields match and the entry is valid, the PTE is considered a hit if the H bit is set as follows:

- If this is the primary PTEG, H = 0
- If this is the secondary PTEG, H = 1

The physical address of the PTE(s) to be checked is derived as shown in Figure 7-22 and Figure 7-23, and the generated address is the address of a group of eight PTEs (a PTEG). During a table search operation, the processor compares up to 16 PTEs: PTE0–PTE7 of the primary PTEG (defined by the primary hashing function) and PTE0–PTE7 of the secondary PTEG (defined by the secondary hashing function).

If the VSID and API fields do not match (or if V or H are not set appropriately) for any of these PTEs, a page fault occurs and an interrupt is taken. Thus, if a valid PTE is located in the page tables, the page is considered resident; if no matching (and valid) PTE is found for an access, the page in question is interpreted as nonresident (page fault) and the operating system must load the page into main memory and update the PTE accordingly.

The architecture does not specify the order in which the PTEs are checked. Note that for maximum performance however, PTEs should be allocated by the operating system first beginning with the PTE0 location within the primary PTEG, then PTE1, and so on. If more than eight PTEs are required within the address space that defines a PTEG address, the secondary PTEG can be used (again, allocation of PTE0 of the secondary PTEG first, and so on is recommended). Additionally, it may be desirable to place the PTEs that will require most frequent access at the beginning of a PTEG and reserve the PTEs in the secondary PTEG for the least frequently accessed PTEs.

The architecture also allows for multiple matching entries to be found within a table search operation. Multiple matching PTEs are allowed if they meet the match criteria described above, as well as have identical RPN, WIMG, and PP values, allowing for differences in the R and C bits. In this case, one of the matching PTEs is used and the R and C bits are updated according to this PTE. In the case that multiple PTEs are found that meet the match criteria but differ in the RPN, WIMG or PP fields, the translation is undefined and the resultant R and C bits in the matching entries are also undefined.

Note that multiple matching entries can also differ in the setting of the H bit, but the H bit must be set according to whether the PTE was located in the primary or secondary PTEG, as described above.

7.7.1.6 Page Table Structure Examples

Figure 7-23 shows the structure of an example page table. The page table base address is defined by SDR1[HTABORG] concatenated with 16 zero bits. In this example, the address is identified by bits 0–13 in SDR1[HTABORG]; note that bits 14 and 15 of HTABORG must be zero because the low-order two bits of HTABMASK are ones. The addresses for individual PTEGs within this page table are then defined by bits 14–25 as an offset from bits 0–13 of this base address. Thus, the page table is defined as 4096 PTEGs.

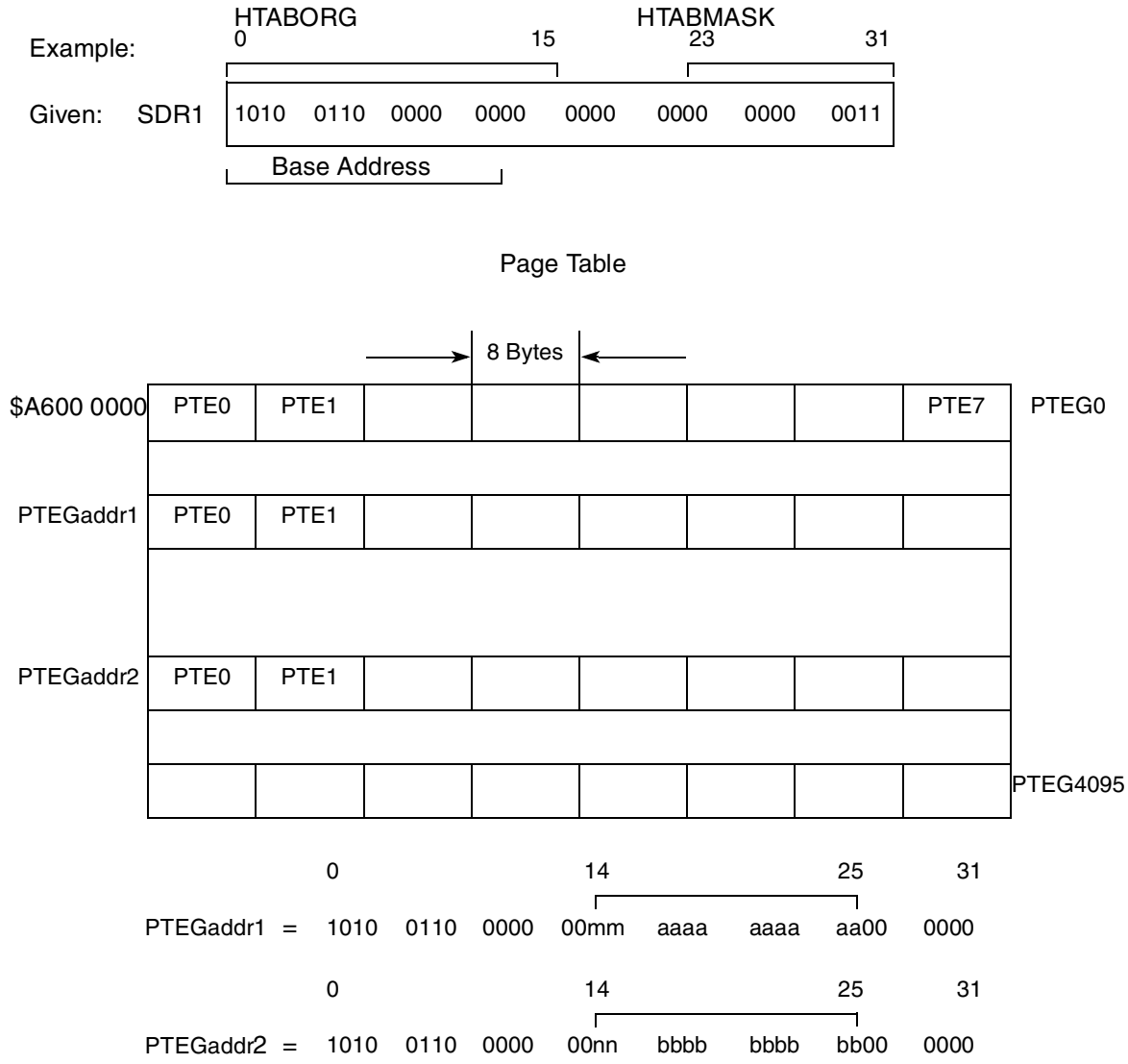


Figure 7-23. Example Page Table Structure

Two example PTEG addresses are shown in the figure as PTEGaddr1 and PTEGaddr2. Bits 14–25 of each PTEG address in this example page table are derived from the output of the hashing function (bits 26–31 are zero to start with PTE0 of the PTEG). In this example, the ‘b’ bits in PTEGaddr2 are the one’s complement of the ‘a’ bits in PTEGaddr1. The ‘n’ bits are also the one’s complement of the ‘m’ bits, but these two bits are generated from bits 7–8 of the output of the hashing function, logically ORed with bits 14–15 of the HTABORG field (which must be zero). If bits 14–25 of PTEGaddr1 were derived by using the primary hashing function, then PTEGaddr2 corresponds to the secondary PTEG.

Note, however, that bits 14–25 in PTEGaddr2 can also be derived from a combination of effective address bits, SR bits, and the primary hashing function. In this case, then PTEGaddr1 corresponds to the secondary

PTEG. Thus, while a PTEG may be considered a primary PTEG for some effective addresses (and SR bits), it may also correspond to the secondary PTEG for a different effective address (and SR value).

It is the value of the H bit in each of the individual PTEs that identifies a particular PTE as either primary or secondary (there may be PTEs that correspond to a primary PTEG and PTEs that correspond to a secondary PTEG, all within the same physical PTEG address space). Thus, only the PTEs that have H = 0 are checked for a hit during a primary PTEG search. Likewise, only PTEs with H = 1 are checked in the case of a secondary PTEG search.

7.7.1.7 PTEG Address Mapping Examples

This section contains two examples of an effective address and how its address translation (the PTE) maps into the primary PTEG in physical memory. The examples illustrate how the processor generates PTEG addresses for a table search operation; this is also the algorithm that must be used by the operating system in creating page tables.

Figure 7-24 shows an example of PTEG address generation. In the example, the value in SDR1 defines a page table at address 0x0F98_0000 that contains 8192 PTEGs. The example effective address selects segment register 0 (SR0) with the 4 highest order bits. The SR0 contents are then used with EA[4–31] to create the 52-bit virtual address.

To generate the address of the primary PTEG, bits 5–23, and bits 24–39 of the virtual address are then used as inputs into the primary hashing function (XOR) to generate hash value 1. The low-order 13 bits of hash value 1 are then concatenated with the high-order 13 bits of HTABORG and with six low-order 0 bits, defining the address of the primary PTEG (0x0F9F_F980).

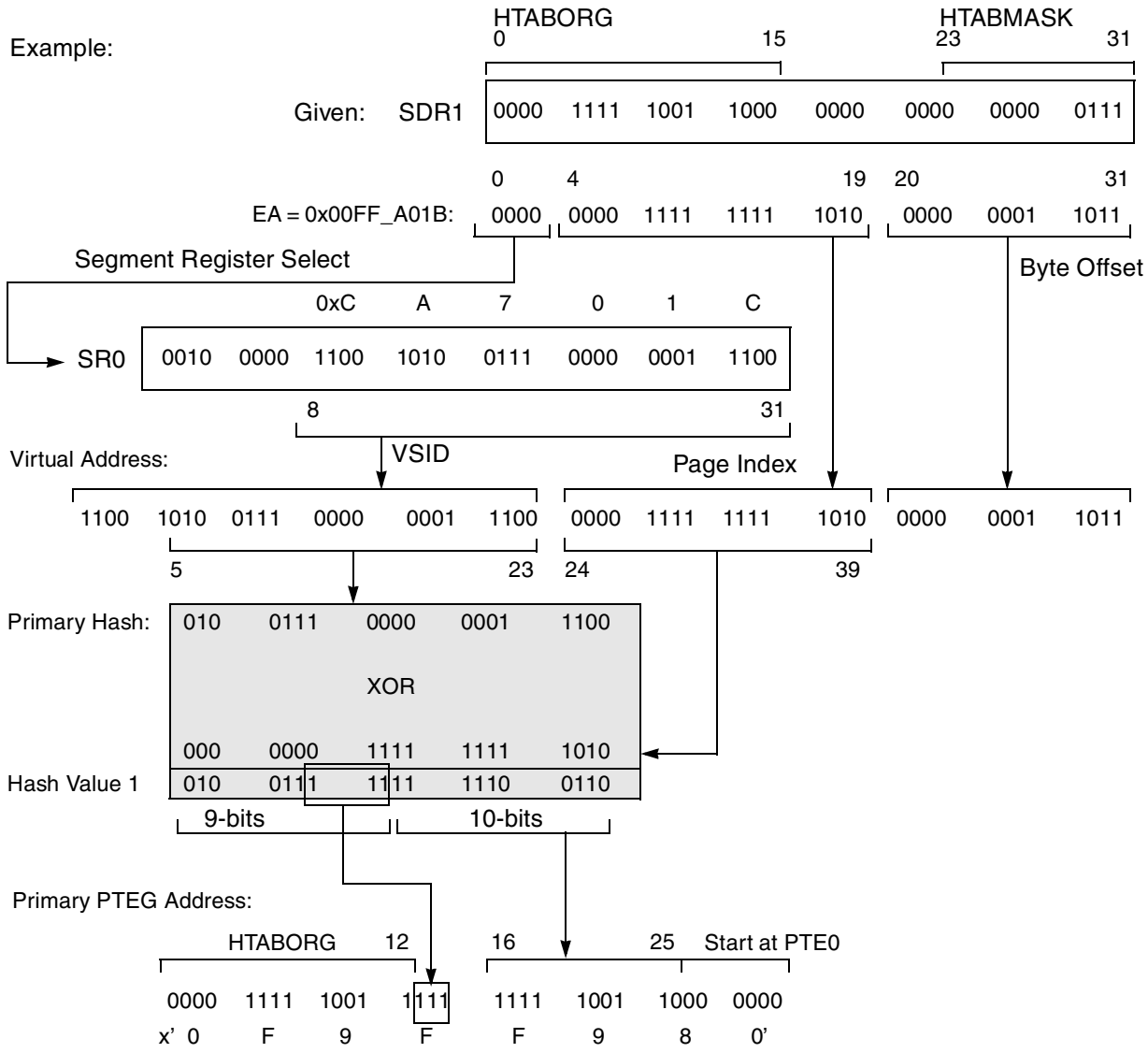


Figure 7-24. Example Primary PTEG Address Generation

Figure 7-25 shows the generation of the secondary PTEG address for this example. If the secondary PTEG is required, the secondary hash function is performed and the low-order 13 bits of hash value 2 are then ORed with the high-order 16 bits of HTABORG (bits 13–15 should be zero), and concatenated with six low-order 0 bits, defining the address of the secondary PTEG (0x0F98_0640).

As described in Figure 7-22, the 10 low-order bits of the page index field are always used in the generation of a PTEG address (through the hashing function). This is why only the abbreviated page index (API) is defined for a PTE (the entire page index field does not need to be checked). For a given effective address, the low-order 10 bits of the page index (at least) contribute to the PTEG address (both primary and secondary) where the corresponding PTE may reside in memory. Therefore, if the high-order 6 bits (the API field) of the page index match with the API field of a PTE within the specified PTEG, the PTE mapping is guaranteed to be the unique PTE required.

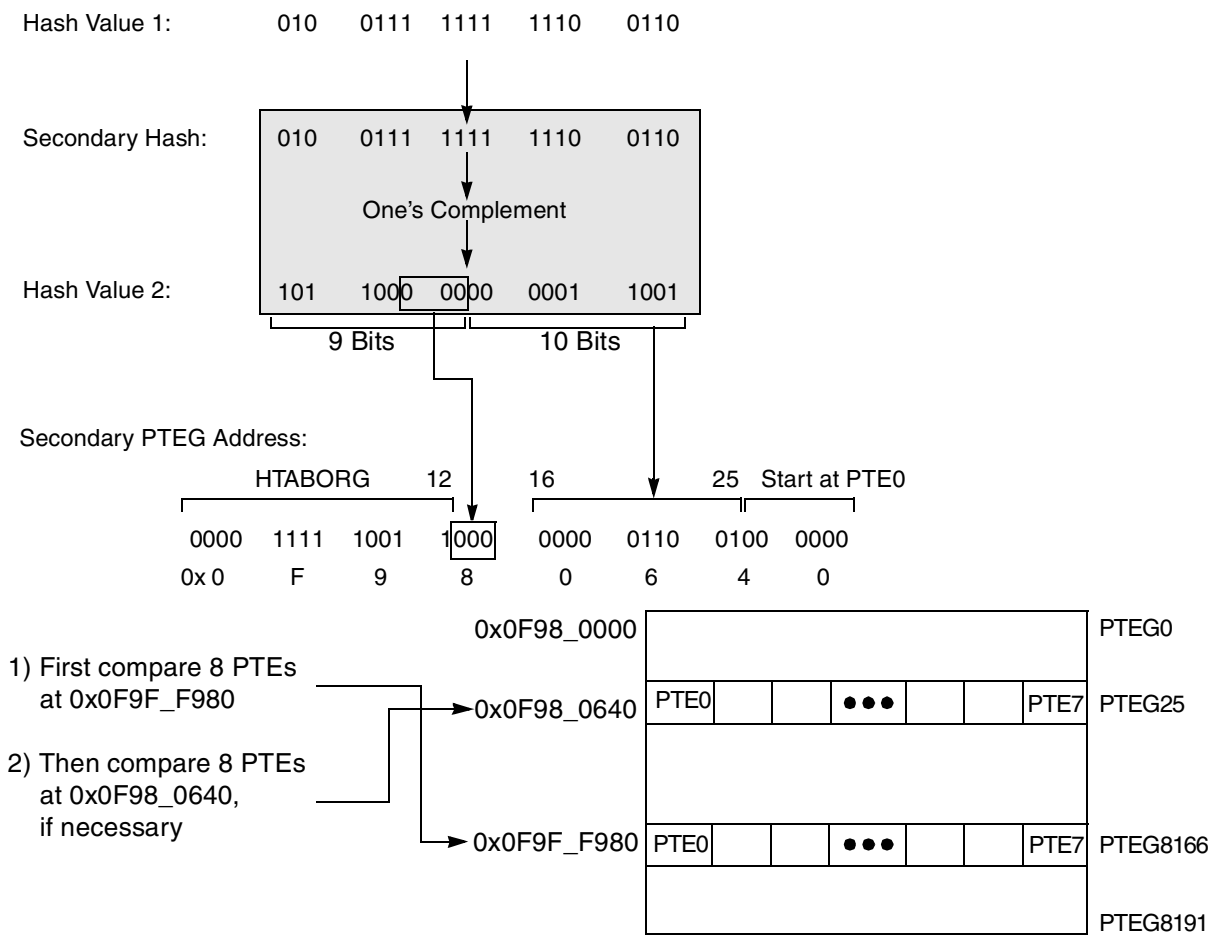


Figure 7-25. Example Secondary PTEG Address Generation

Note that a given PTEG address does not map back to a unique effective address. Not only can a given PTEG be considered both a primary and a secondary PTEG (as described in Section 7.7.1.6, "Page Table Structure Examples"), but in this example, bits 24–26 of the page index field of the virtual address are not used to generate the PTEG address. Therefore, any of the eight combinations of these bits will map to the same primary PTEG address. (However, these bits are part of the API and are therefore compared for each PTE within the PTEG to determine if there is a hit.) Furthermore, an effective address can select a different SR with a different value such that the output of the primary (or secondary) hashing function happens to equal the hash values shown in the example. Thus, these effective addresses would also map to the same PTEG addresses shown.

7.7.2 Page Table Search Operation

An outline of the page table search process performed is as follows:

1. The 32-bit physical addresses of the primary and secondary PTEGs are generated as described in Section 7.7.1.4, “Page Table Addresses.”
2. As many as 16 PTEs (from the primary and secondary PTEGs) are read from memory (the architecture does not specify the order of these reads, allowing multiple reads to occur in parallel). PTE reads occur with an implied WIM memory/cache mode control bit setting of 0b001. Therefore, they are considered cacheable.
3. The PTEs in the selected PTEGs are tested for a match with the virtual page number (VPN) of the access. The VPN is the VSID concatenated with the page index field of the virtual address. For a match to occur, the following must be true:
   — PTE[H] = 0 for primary PTEG; PTE[H] = 1 for secondary PTEG
   — PTE[V] = 1
   — PTE[VSID] = VA[0–23]
   — PTE[API] = VA[24–29]
4. If a match is not found within the eight PTEs of the primary PTEG and the eight PTEs of the secondary PTEG, an interrupt is generated as described in step 8. If a match (or multiple matches) is found, the table search process continues.
5. If multiple matches are found, all of the following must be true:
   — PTE[RPN] is equal for all matching entries
   — PTE[WIMG] is equal for all matching entries
   — PTE[PP] is equal for all matching entries
6. If one of the fields in step 5 does not match, the translation is undefined, and R and C bit of matching entries are undefined. Otherwise, the R and C bits are updated based on one of the matching entries.
7. A copy of the PTE is written into the on-chip TLB (if implemented) and the R bit is updated in the PTE in memory (if necessary). If there is no memory protection violation, the C bit is also updated in memory (if necessary) and the table search is complete.
8. If a match is not found within the primary or secondary PTEG, the search fails, and a page fault exception occurs (causing either an ISI or DSI).

Reads from memory for page table search operations are performed (that is, as unguarded cacheable operations in which coherency is required).

7.7.2.1 Flow for Page Table Search Operation

Figure 7-26 provides a detailed flow diagram of a page table search operation. Note that the references to TLBs are shown as optional because TLBs are not required; if they do exist, the specifics of how they are maintained are implementation-specific. Also, Figure 7-26 shows only a few cases of R-bit and C-bit updates. For a complete list of the R- and C-bit updates dictated by the architecture, refer to Table 7-14.

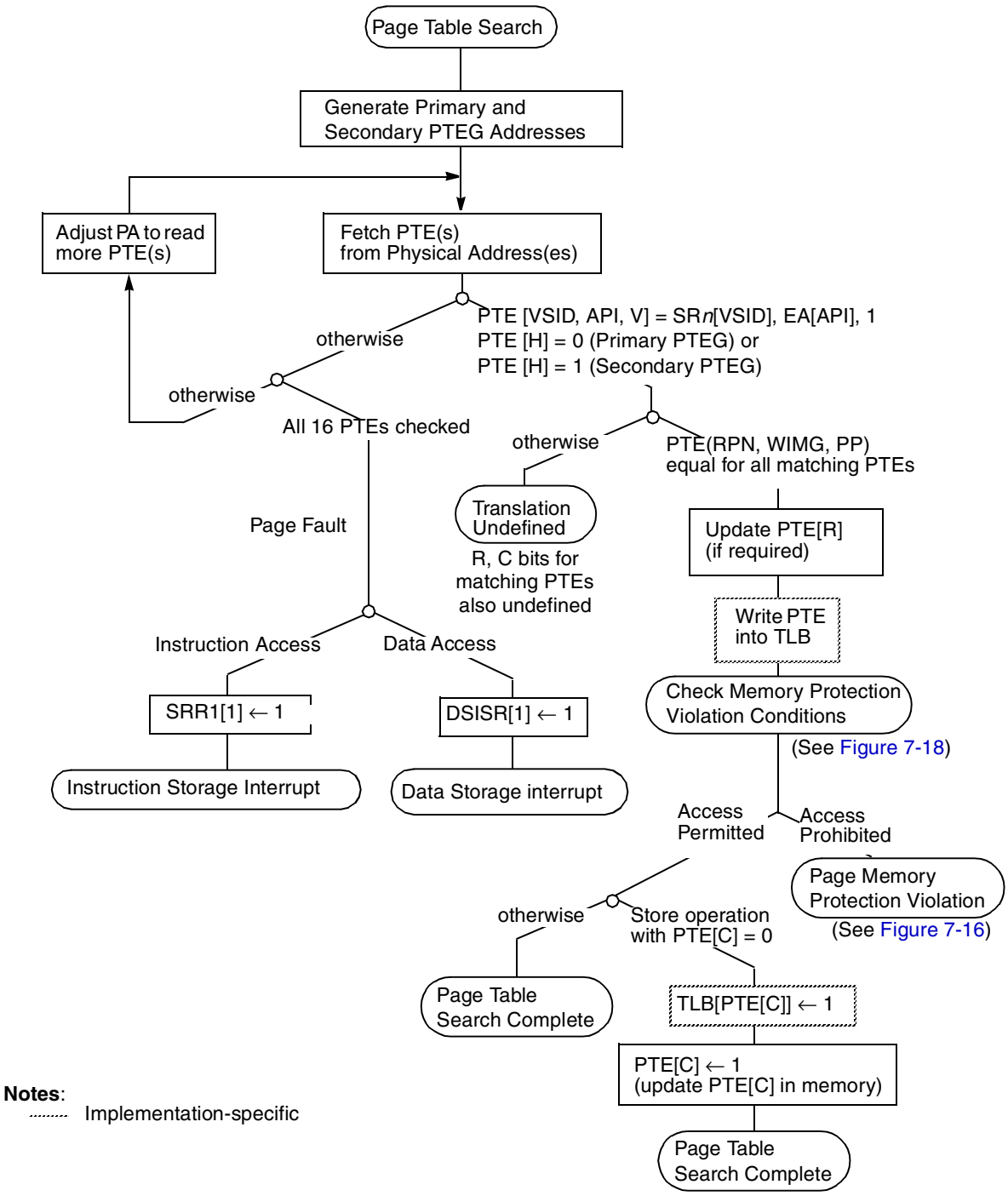


Figure 7-26. Page Table Search Flow

7.7.3 Page Table Updates

This section describes the requirements on the software when updating page tables in memory via some pseudocode examples. Multiprocessor systems must follow the rules described in this section so that all processors operate with a consistent set of page tables. Even single processor systems must follow certain

rules, because software changes must be synchronized with the other instructions in execution and with automatic updates that may be made by the hardware (reference and change bit updates). Updates to the tables include the following operations:

- Adding a PTE
- Deleting a PTE

PTEs must be locked on multiprocessor systems. Access to PTEs must be appropriately synchronized by software locking of (that is, guaranteeing exclusive access to) PTEs or PTEGs if more than one processor can modify the table at that time. In the examples below, software locks should be performed to provide exclusive access to the PTE being updated. However, the architecture does not dictate the specific protocol to be used for locking (for example, a single lock, a lock per PTEG, or a lock per PTE can be used). See Appendix D, "Synchronization Programming Examples," for more information about the use of the reservation instructions (such as the **lwarx** and **stwcx.** instructions) to perform software locking.

TLBs are implemented as noncoherent caches of the page tables. TLB entries must be invalidated explicitly with the TLB invalidate entry instruction (**tlbie**) whenever the corresponding PTE is modified. In a multiprocessor system, the **tlbie** instruction must be controlled by software locking, so that the **tlbie** is issued on only one processor at a time.

The OEA defines the **tlbsync** instruction that ensures that TLB invalidate operations executed by this processor have caused all appropriate actions in other processors. In a system that contains multiple processors, the **tlbsync** functionality must be used to ensure proper synchronization with the other processors. Note that a **sync** instruction must also follow the **tlbsync** to ensure that the **tlbsync** has completed execution on this processor.

On single processor systems, PTEs need not be locked and the **eieio** instructions (in between the **tlbie** and **tlbsync** instructions) and the **tlbsync** instructions themselves are not required. The **sync** instructions shown are required even for single processor systems (to ensure that all previous changes to the page tables and all preceding **tlbie** instructions have completed).

Any processor, including the processor modifying the page table, may access the page table at any time in an attempt to reload a TLB entry. An inconsistent PTE must never accidentally become visible (if V = 1); thus, there must be synchronization between modifications to the valid bit and any other modifications (to avoid corrupted data).

In the pseudocode examples that follow, changes made to a PTE shown as a single line in the example are assumed to be performed with an atomic store instruction. Appropriate modifications must be made to these examples if this assumption is not satisfied.

Updates of R and C bits by the processor are not synchronized with the accesses that cause the updates. When modifying the low-order half of a PTE, software must take care to avoid overwriting a processor update of these bits and to avoid having the value written by a store instruction overwritten by a processor update. The processor does not alter any other fields of the PTE.

For a complete list of the synchronization requirements for executing the MMU instructions, see Section 2.3.17, "Synchronization Requirements for Special Registers and for Lookaside Buffers."

The following examples show the required sequence of operations. However, other instructions may be interleaved within the sequences shown. Page tables are modified by deleting and adding a page table entry.

7.7.3.1 Adding a Page Table Entry

Adding a page table entry requires only a lock on the PTE in a multiprocessor system. The first bytes in the PTE are then written (this example assumes the old valid bit was cleared), the **eieio** instruction orders the update, and then the second update can be made. A **sync** instruction ensures that the updates have been made to memory.

```
lock(PTE)
PTE[RPN,R,C,WIMG,PP] ← new values
eieio   /* order 1st PTE update before 2nd
PTE[VSID,H,API,V] ← new values (V = 1)
sync    /* ensure updates completed
unlock(PTE)
```

7.7.3.2 Deleting a Page Table Entry

In this example, the entry is locked, marked invalid, invalidated in the TLB, and unlocked.

Again, note that the **tlbsync** and the **sync** instruction that follows it are only required if consistency must be maintained with other processors in a multiprocessor system (and the software is to be used in a multiprocessor environment).

```
lock(PTE)
PTE[V] ← 0   /* (other fields don't matter)
sync         /* ensure update completed
tlbie(old_EA) /* invalidate old translation
eieio        /* order tlbie before tlbsync
tlbsync      /* ensure tlbie completed on all processors
sync         /* ensure tlbsync completed
unlock(PTE)
```

7.7.4 Segment Register Updates

Synchronization requirements for using the move to segment register instructions are described in Section 2.3.17, "Synchronization Requirements for Special Registers and for Lookaside Buffers."

7.8 Direct-Store Segment Address Translation

As described for memory segments, all accesses generated by the processor (with translation enabled) that do not map to a BAT area, map to an SR. If SR[T] = 1, the access maps to the direct-store interface, invoking a specific bus protocol for accessing I/O devices.

Direct-store segments are provided for POWER compatibility. As the direct-store interface is present only for compatibility with existing I/O devices that used this interface and the direct-store interface protocol is not optimized for performance, its use is discouraged. This functionality is considered optional (to allow for those earlier devices that implemented it). However, future devices are not likely to support it. Thus, software should not depend on its results and new software should not use it. Applications that require

low-latency load/store access to external address space should use memory-mapped I/O, rather than the direct-store interface.

7.8.1 Segment Registers for Direct-Store Segments

The format of many SR fields depends on the value of SR[T]. Figure 7-27 shows the SR format when SR[T] is set.

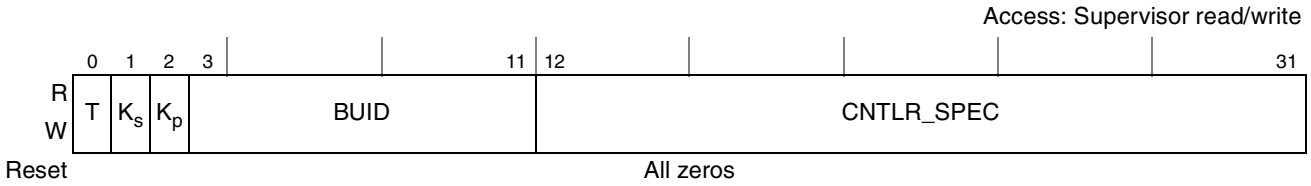


Figure 7-27. Segment Register Format for Direct-Store Segments

Table 7-20 shows the SR field definitions when the T bit is set.

Table 7-20. Segment Register Bit Definitions for Direct-Store Segments

Bit	Name	Description
0	T	T = 1 selects this format.
1	Ks	Supervisor-state protection key
2	Kp	User-state protection key
3–11	BUID	Bus unit ID
12–31	CNTLR_SPEC	Device-specific data for I/O controller

7.8.2 Direct-Store Segment Accesses

When the address translation process determines that SR[T] = 1, direct-store segment address translation is selected; no reference is made to the page tables and neither the reference or change bits are updated. These accesses are performed as if the WIMG bits were 0b0101; that is, caching is inhibited, the accesses bypass the cache, hardware-enforced coherency is not required, and the accesses are considered guarded.

The specific protocol invoked to perform these accesses involves the transfer of address and data information; however, the OEA does not define the exact hardware protocol used for direct-store accesses. Some instructions may cause multiple address/data transactions to occur on the bus. In this case, the address for each transaction is handled individually with respect to the MMU.

The following describes the data that is typically sent to the memory controller by processors that implement the direct-store function:

- One of the K*x* bits (Ks or Kp) is selected to be the key as follows:
  - For supervisor accesses (MSR[PR] = 0), the Ks bit is used and Kp is ignored.
  - For user accesses (MSR[PR] = 1), the Kp bit is used and Ks is ignored.
- An implementation-dependent portion of the SR.
- An implementation-dependent portion of the effective address.

7.8.3 Direct-Store Segment Protection

Page-level memory protection as described in Section 7.6.4, “Page Memory Protection,” is not provided for direct-store segments. The appropriate key bit (Ks or Kp) from the segment register is sent to the memory controller, and the memory controller implements any protection required. Frequently, no such mechanism is provided; the fact that a direct-store segment is mapped into the address space of a process may be regarded as sufficient authority to access the segment.

7.8.4 Instructions Not Supported in Direct-Store Segments

The following instructions are not supported and cause either a DSI interrupt or boundedly-undefined results when issued with an effective address for which SR*n*[T] = 1: **lwarx**, **stwcx.**, **eciwx**, and **ecowx**.

7.8.5 Instructions with No Effect in Direct-Store Segments

The following instructions are executed as no-ops when issued with an effective address that selects a segment where T = 1: **dcba**, **dcbt**, **dcbtst**, **dcbf**, **dcbi**, **dcbst**, **dcbz**, and **icbi**

7.8.6 Direct-Store Segment Translation Summary Flow

Figure 7-28 shows the flow used by the MMU when direct-store segment address translation is selected. This figure expands the Direct-Store Segment Translation stub found in Figure 7-4 for both instruction and data accesses. In the case of a floating-point load or store operation to a direct-store segment, it is implementation-specific whether the alignment interrupt occurs. In the case of an **eciwx**, **ecowx**, **lwarx**, or **stwcx.** instruction, the implementation either sets the DSISR as shown and causes the DSI, or causes boundedly-undefined results.

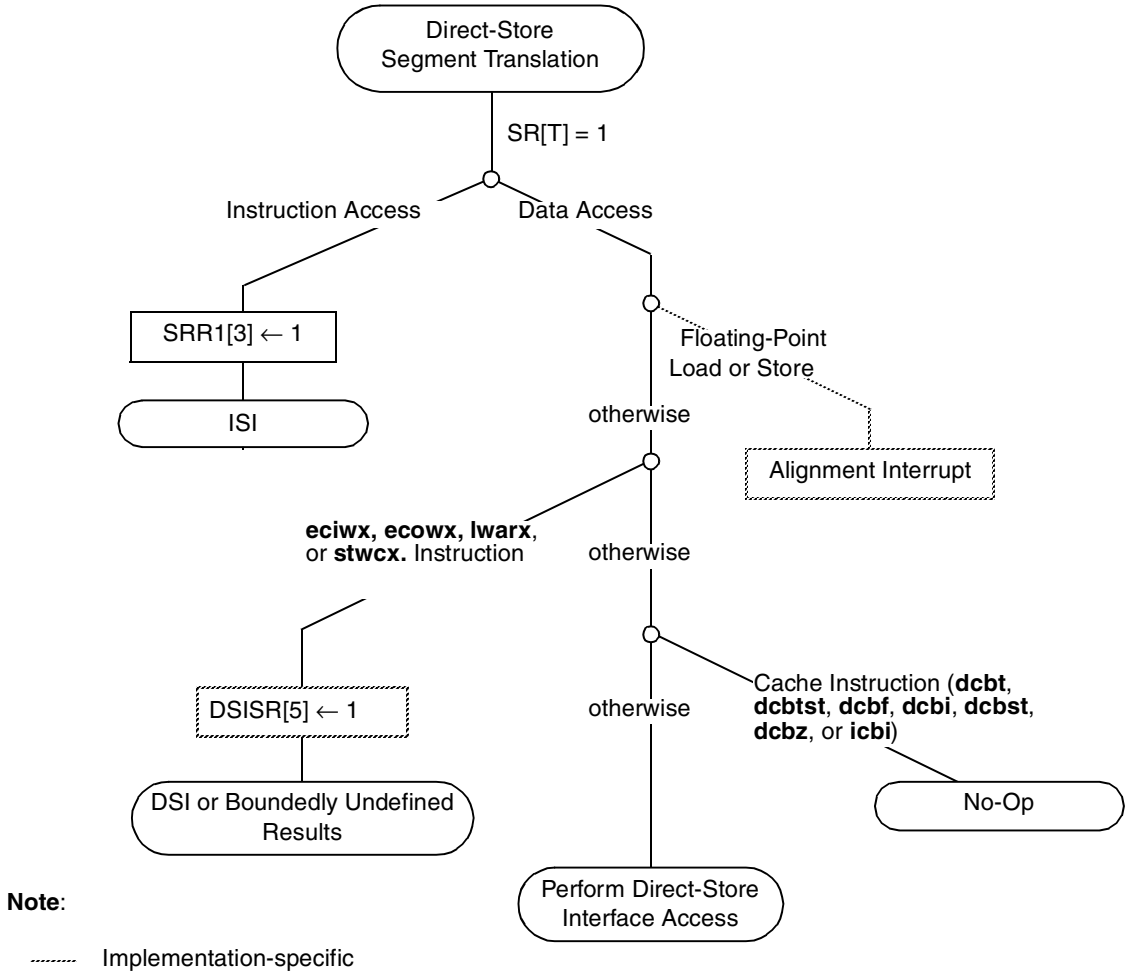


Figure 7-28. Direct-Store Segment Translation Flow

Chapter 8
Instruction Set

This chapter lists instructions in alphabetical order by mnemonic. Each entry includes the instruction formats and a legend that shows the level or levels of the architecture in which the instruction may be found (UISA, VEA, or OEA); the privilege level (user or supervisor); and the instruction formats. The format diagrams show all valid combinations of instruction fields. The legend also indicates if the instruction is 32-bit, and whether it is optional. Chapter 4, "Addressing Modes and Instruction Set Summary," gives a higher-level description of the instruction set.

8.1 Instruction Formats

Instructions are 4 bytes long and word-aligned, so when instruction addresses are presented to the processor (as in branch instructions) the 2 low-order bits are ignored. Similarly, whenever the processor develops an instruction address, its 2 low-order bits are zero. Bits 0–5 always specify the primary opcode. Many instructions also have an extended opcode. The remaining bits contain one or more fields for the different instruction formats.

Some instruction fields are reserved or must contain a predefined value. If a reserved field does not have all bits cleared or if a field that must contain a particular value does not contain that value, the instruction form is invalid and the results are as described in Chapter 4, "Addressing Modes and Instruction Set Summary."

8.1.1 Split-Field Notation

Split fields occupy more than one contiguous sequence of bits or occupy a contiguous sequence of bits used in permuted order. Table 8-1 describes split fields that represent the concatenation of the sequences from left to right. They are shown in lowercase letters (spr and tbr). Split fields that represent the concatenation of the sequences in some order, which need not be left to right (as described for each affected instruction), are shown in uppercase letters. Table 8-2 describes these split fields: MB, ME, and SH.

Table 8-1. Split-Field Notation and Conventions

Field	Description
spr (11:20)	Specifies an spr for **mtspr** and **mfspr** instructions.
tbr (11:20)	Specifies either the time base lower (TBL) or time base upper (TBU).

8.1.2 Instruction Fields

Table 8-2 describes the instruction fields used in the various instruction formats.

Table 8-2. Instruction Syntax Conventions

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address (CIA). (For more information on th CIA, see Table 8-3.) The effective (logical) address of the branch is either the sum of the LI field sign-extended to 32 bits and the address of the branch instruction or the sum of the BD field sign-extended to 32 bits and the address of the branch instruction. 1 The immediate field represents an absolute address. The effective address (EA) of the branch is the LI field sign-extended to 32 bits or the BD field sign-extended to 32 bits.
BD (16:29)	Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 32 bits.
BI (11:15)	Used to specify a CR bit to be used as the condition of a branch conditional instruction.
BO (6:10)	Used to specify options for the branch conditional instructions. The encoding is described in Section 4.2.4.2, “Conditional Branch Control.”
crbA (11:15)	Used to specify a CR bit to be used as a source.
crbB (16:20)	Used to specify a CR bit to be used as a source.
crbD (6:10)	Used to specify a CR bit, or in the FPSCR, as the destination of the result of an instruction.
crfD (6:8)	Used to specify one of the CR fields, or one of the FPSCR fields, as a destination.
crfS (11:13)	Used to specify one of the CR fields, or one of the FPSCR fields, as a source.
CRM (12:19)	This field mask is used to identify the CR fields that are to be updated by the **mtcrf** instruction.
d (16:31)	Immediate field specifying a 16-bit signed two's complement integer that is sign-extended to 32 bits.
FM (7:14)	This field mask is used to identify the FPSCR fields that are to be updated by the **mtfsf** instruction.
frA (11:15)	Used to specify an FPR as a source.
frB (16:20)	Used to specify an FPR as a source.
frC (21:25)	Used to specify an FPR as a source.
frD (6:10)	Used to specify an FPR as the destination.
frS (6:10)	Used to specify an FPR as a source.
IMM (16:19)	Immediate field used as the data to be placed into a field in the FPSCR.
LI (6:29)	Immediate field specifying a 24-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 32 bits.
LK (31)	Link bit. 0 Does not update the link register (LR). 1 Updates the LR. If the instruction is a branch instruction, the address of the instruction following the branch instruction is placed into the LR.

Table 8-2. Instruction Syntax Conventions (continued)

Field	Description
MB (21:25) and ME (26:30)	Used in rotate instructions to specify a 32-bit mask as described in Section 4.2.1.4, "Integer Rotate and Shift Instructions."
NB (16:20)	Used to specify the number of bytes to move in an immediate string load or store.
OE (21)	Used for extended arithmetic to enable setting OV and SO in the XER.
OPCD (0:5)	Primary opcode field
rA (11:15)	Used to specify a GPR to be used as a source or destination.
rB (16:20)	Used to specify a GPR to be used as a source.
Rc (31)	Record bit. 0 Does not update the condition register (CR). 1 Updates the CR to reflect the result of the operation. For integer instructions, CR[0:2] reflects the result as a signed quantity and CR[3] receives a copy of the summary overflow bit, XER[SO]. The result as an unsigned quantity or a bit string can be deduced from the EQ bit. For floating-point instructions, CR[4:7] reflect floating-point, floating-point enabled, floating-point invalid operation, and floating-point overflow exceptions.
rD (6:10)	Used to specify a GPR to be used as a destination.
rS (6:10)	Used to specify a GPR to be used as a source.
SH (16:20)	Used to specify a shift amount.
SIMM (16:31)	This immediate field is used to specify a 16-bit signed integer.
SR (12:15)	Used to specify one of the 16 segment registers.
TO (6:10)	Specifies conditions on which to trap. The encoding is described in Section 4.2.4.6, "Trap Instructions."
UIMM (16:31)	This immediate field is used to specify a 16-bit unsigned integer.
XO (21:30, 22:30, 26:30)	Extended opcode field.

8.1.3 Notation and Conventions

The operation of some instructions is described by a semiformal language (pseudocode). Table 8-3 describes pseudocode notation and conventions.

Table 8-3. Notation and Conventions

Notation/Convention	Meaning
←	Assignment
←iea	Assignment of an instruction effective address.
¬	NOT logical operator
*	Multiplication

Table 8-3. Notation and Conventions (continued)

Notation/Convention	Meaning
÷	Division (yielding quotient)
+	Two's-complement addition
–	Two's-complement subtraction, unary minus
=, ≠	Equals and Not Equals relations
<, ≤, >, ≥	Signed comparison relations
. (period)	Update. When used as a character of an instruction mnemonic, a period (.) means that the instruction updates the condition register field.
c	Carry. When used as a character of an instruction mnemonic, a 'c' indicates a carry out in XER[CA].
e	Extended Precision. When used as the last character of an instruction mnemonic, an 'e' indicates the use of XER[CA] as an operand in the instruction and records a carry out in XER[CA].
o	Overflow. When used as a character of an instruction mnemonic, an 'o' indicates the record of an overflow in XER[OV] and CR0[SO] for integer instructions or CR1[SO] for floating-point instructions.
<U, >U	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
||	Used to describe the concatenation of two values (that is, 010 || 111 is the same as 010111)
⊕, ≡	Exclusive-OR, Equivalence logical operators (for example, (a ≡ b) = (a ⊕ ¬ b))
0b*nnnn*	A number expressed in binary format.
0x*nnnn*	A number expressed in hexadecimal format.
(*n*)x	The replication of x, *n* times (that is, x concatenated to itself *n* – 1 times). (*n*)0 and (*n*)1 are special cases. A description of the special cases follows: • (*n*)0 means a field of *n* bits with each bit equal to 0. Thus (5)0 is equivalent to 0b00000. • (*n*)1 means a field of *n* bits with each bit equal to 1. Thus (5)1 is equivalent to 0b11111.
(**r**A|0)	The contents of **r**A if the **r**A field has the value 1–31, or the value 0 if the **r**A field is 0.
(**r**X)	The contents of **r**X
x[*n*]	*n* is a bit or field within x, where x is a register
x^n	x is raised to the *n*th power
ABS(x)	Absolute value of x
CEIL(x)	Least integer ≥ x
Characterization	Reference to the setting of status bits in a standard way that is explained in the text.
CIA	Current instruction address. The 32-bit address of the instruction being described by a sequence of pseudocode. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK = 1 to set the link register. Does not correspond to any architected register.
Clear	Clear the leftmost or rightmost *n* bits of a register (to 0). This operation is used for rotate and shift instructions.

Table 8-3. Notation and Conventions (continued)

Notation/Convention	Meaning
Clear left and shift left	Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a known non-negative array index by the width of an element. These operations are used for rotate and shift instructions.
Cleared	Bits are set to 0.
Do	Do loop. • Indenting shows range. • “To” and/or “by” clauses specify incrementing an iteration variable. • “While” clauses give termination conditions.
DOUBLE(x)	Result of converting x from floating-point single-precision to floating-point double-precision format.
Extract	Select a field of *n* bits starting at bit position *b* in the source register, right or left justify this field in the target register, and clear all other bits of the target register to zero. This operation is used for rotate and shift instructions.
EXTS(x)	Result of extending x on the left with sign bits
GPR(x)	General-purpose register x
if...then...else...	Conditional execution, indenting shows range, else is optional.
Insert	Select a field of *n* bits in the source register, insert this field starting at bit position *b* of the target register, and leave other bits of the target register unchanged. (No simplified mnemonic is provided for insertion of a field when operating on double words; such an insertion requires more than one instruction.) This operation is used for rotate and shift instructions. (Note that simplified mnemonics are referred to as extended mnemonics in the architecture specification.)
Leave	Leave innermost do loop, or the do loop described in leave statement.
MASK(x, y)	Mask having ones in positions x through y (wrapping if x > y) and zeros elsewhere.
MEM(x, y)	Contents of y bytes of memory starting at address x.
NIA	Next instruction address, which is the32-bit address of the next instruction to be executed (the branch destination) after a successful branch. In pseudocode, a successful branch is indicated by assigning a value to NIA. For instructions which do not branch, the next instruction address is CIA + 4. Does not correspond to any architected register.
OEA	Operating environment architecture
Rotate	Rotate the contents of a register right or left *n* bits without masking. This operation is used for rotate and shift instructions.
ROTL[64](x, y)	Result of rotating the 64-bit value x left y positions
ROTL[32](x, y)	Result of rotating the 64-bit value x || x left y positions, where x is 32 bits long
Set	Bits are set (to 1).
Shift	Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift). This operation is used for rotate and shift instructions.
SINGLE(x)	Result of converting x from floating-point double-precision format to floating-point single-precision format.
SPR(x)	Special-purpose register x
TRAP	Invoke the system trap handler.

Table 8-3. Notation and Conventions (continued)

Notation/Convention	Meaning
Undefined	An undefined value. The value may vary from one implementation to another, and from one execution to another on the same implementation.
UISA	User instruction set architecture
VEA	Virtual environment architecture

Table 8-4 describes instruction field notation conventions used throughout this chapter.

Table 8-4. Instruction Field Conventions

Architecture Specification	Equivalent to:
BA, BB, BT	**crb**A, **crb**B, **crb**D (respectively)
BF, BFA	**crf**D, **crf**S (respectively)
D	d
DS	ds
FLM	FM
FRA, FRB, FRC, FRT, FRS	**fr**A, **fr**B, **fr**C, **fr**D, **fr**S (respectively)
FXM	CRM
RA, RB, RT, RS	**r**A, **r**B, **r**D, **r**S (respectively)
SI	SIMM
U	IMM
UI	UIMM
/, //, ///	0...0 (shaded)

Some operators are applied before others, as shown in Table 8-5. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. For example, “–” (unary minus) associates from left to right, so a – b – c = (a – b) – c. Parentheses are used to override the evaluation order implied by Table 8-5 or to increase clarity; parenthesized expressions are evaluated before serving as operands.

Table 8-5. Precedence Rules

Precedence	Operators	Associativity
Highest	x[*n*], function evaluation	Left to right
	(*n*)x or replication, x(*n*) or exponentiation	Right to left
	unary –, ¬	Right to left
	*, ÷	Left to right
	+, –	Left to right
	||	Left to right
	=, ≠, <, ≤, >, ≥, <U, >U, ?	Left to right
	&, ⊕, ≡	Left to right
	|	Left to right
	: (range)	None
Lowest	←, ←iea	None

8.2 Instruction Set

The remainder of this chapter lists and describes individual instructions, which are listed in alphabetical order by mnemonic. Figure 8-1 shows the format for each instruction description page.

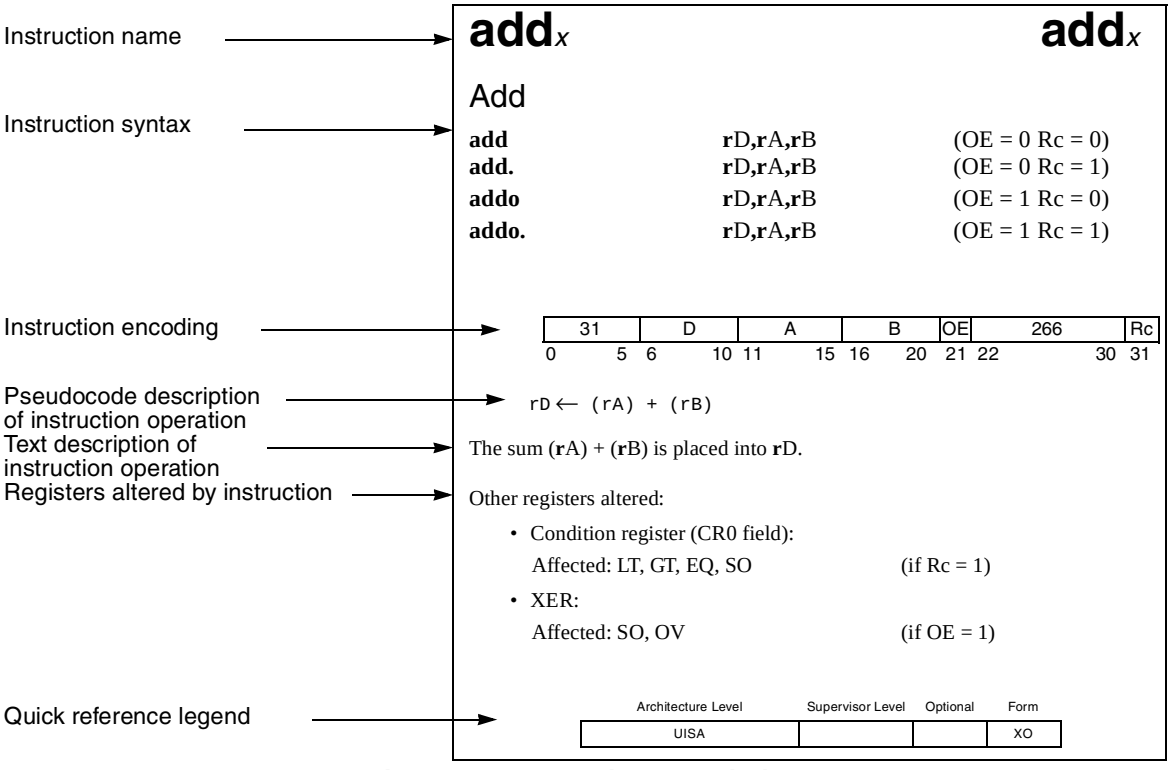


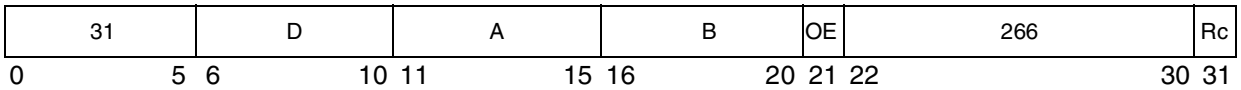
Figure 8-1. Instruction Description

Note that the execution unit that executes the instruction may not be the same for all processors.

add*x* add*x*

Add

add **r**D,**r**A,**r**B (OE = 0 Rc = 0)
add. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
addo **r**D,**r**A,**r**B (OE = 1 Rc = 0)
addo. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
rD← (rA) + (rB)
```

The sum (**r**A) + (**r**B) is placed into **r**D.

The **add** instruction is preferred for addition because it sets few status bits.

Other registers altered:

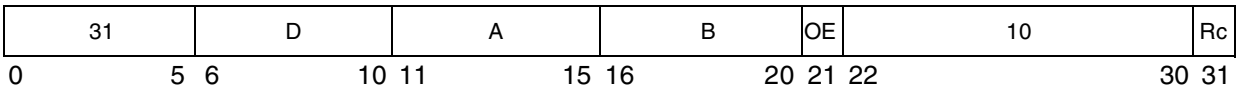
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

addc*x* addc*x*

Add Carrying

addc **r**D,**r**A,**r**B (OE = 0 Rc = 0)
addc. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
addco **r**D,**r**A,**r**B (OE = 1 Rc = 0)
addco. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
rD ← (rA) + (rB)
```

The sum (**r**A) + (**r**B) is placed into **r**D.

Other registers altered:

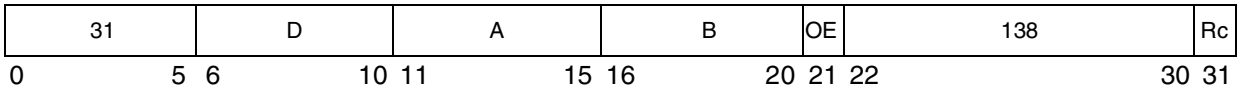
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

adde*x* adde*x*

Add Extended

adde	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
adde.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
addeo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
addeo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
rD ← (rA) + (rB) + XER[CA]
```

The sum (**r**A) + (**r**B) + XER[CA] is placed into **r**D.

Other registers altered:

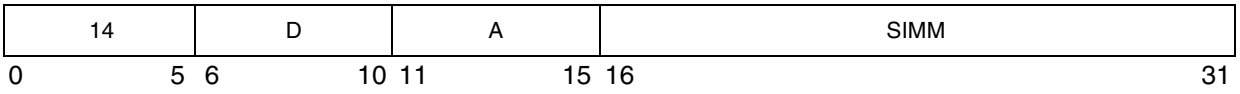
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

addi addi

Add Immediate

addi **r**D**,r**A,SIMM




```
if rA = 0 then rD ← EXTS(SIMM)
else     rD ← rA + EXTS(SIMM)
```

The sum (**r**A|0) + SIMM is placed into **r**D.

The **addi** instruction is preferred for addition because it sets few status bits. Note that **addi** uses the value 0, not the contents of GPR0, if **r**A = 0.

Other registers altered:

- None

Simplified mnemonics:

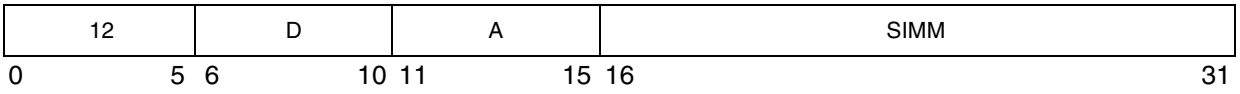
li	**r**D**,**value	equivalent to	**addi**	**r**D**,0,**value
la	**r**D**,**disp(**r**A)	equivalent to	**addi**	**r**D**,r**A**,**disp
subi	**r**D**,r**A**,**value	equivalent to	**addi**	**r**D**,r**A**,**–value

Architecture Level	Supervisor Level	Optional	Form
UISA			D

addic addic

Add Immediate Carrying

addic **r**D,**r**A,SIMM




```
rD ← (rA) + EXTS(SIMM)
```

The sum (**r**A) + SIMM is placed into **r**D.

Other registers altered:

- XER:
  Affected: CA

Simplified mnemonics:

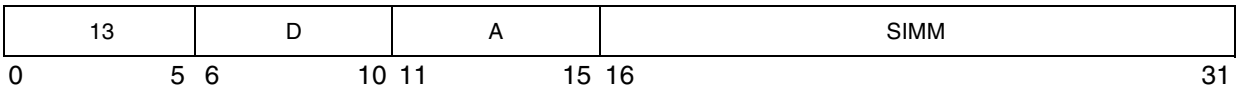
subic **r**D,**r**A,value equivalent to **addic** **r**D,**r**A,–value

Architecture Level	Supervisor Level	Optional	Form
UISA			D

addic. addic.

Add Immediate Carrying and Record

addic. **r**D,**r**A,SIMM




```
rD ← (rA) + EXTS(SIMM)
```

The sum (**r**A) + SIMM is placed into **r**D.

Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA

Simplified mnemonics:

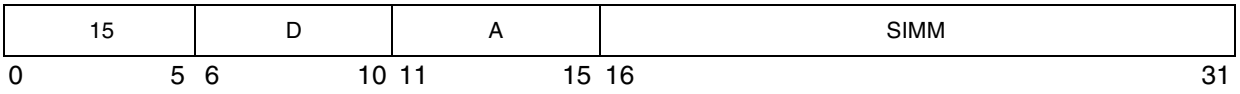
subic. **r**D,**r**A,value equivalent to **addic.** **r**D,**r**A,–value

Architecture Level	Supervisor Level	Optional	Form
UISA			D

addis addis

Add Immediate Shifted

addis **r**D**,r**A,SIMM




```
if rA = 0 then rD ← EXTS(SIMM || (16)0)
else      rD ← (rA) + EXTS(SIMM || (16)0)
```

The sum (**r**A|0) + (SIMM || 0x0000) is placed into **r**D.

The **addis** instruction is preferred for addition because it sets few status bits. Note that **addis** uses the value 0, not the contents of GPR0, if **r**A = 0.

Other registers altered:

- None

Simplified mnemonics:

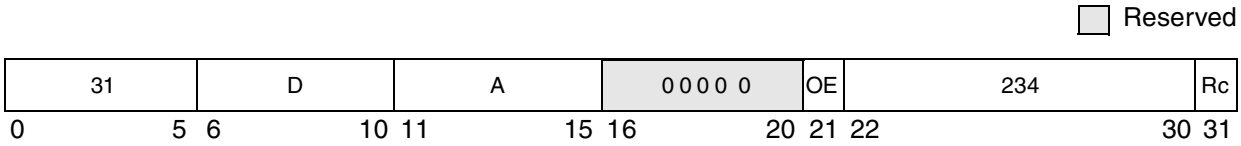
lis	**r**D**,**value	equivalent to	**addis**	**r**D**,0,**value
subis	**r**D**,r**A**,**value	equivalent to	**addis**	**r**D**,r**A**,**–value

Architecture Level	Supervisor Level	Optional	Form
UISA			D

addme*x* addme*x*

Add to Minus One Extended

addme	**r**D,**r**A	(OE = 0 Rc = 0)
addme.	**r**D,**r**A	(OE = 0 Rc = 1)
addmeo	**r**D,**r**A	(OE = 1 Rc = 0)
addmeo.	**r**D,**r**A	(OE = 1 Rc = 1)




```
rD ← (rA) + XER[CA] – 1
```

The sum (**r**A) + XER[CA] + 0xFFFF_FFFF is placed into **r**D.

Other registers altered:

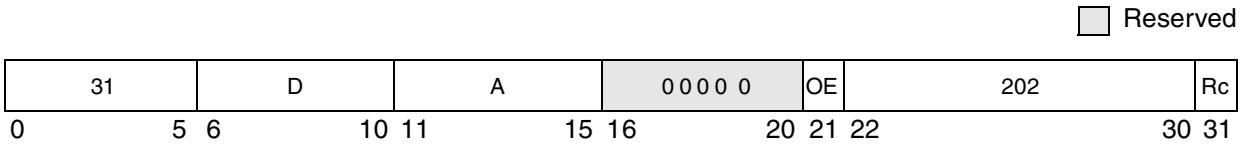
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

addze*x* addze*x*

Add to Zero Extended

addze **r**D,**r**A (OE = 0 Rc = 0)
addze. **r**D,**r**A (OE = 0 Rc = 1)
addzeo **r**D,**r**A (OE = 1 Rc = 0)
addzeo. **r**D,**r**A (OE = 1 Rc = 1)




```
rD ← (rA) + XER[CA]
```

The sum (**r**A) + XER[CA] is placed into **r**D.

Other registers altered:

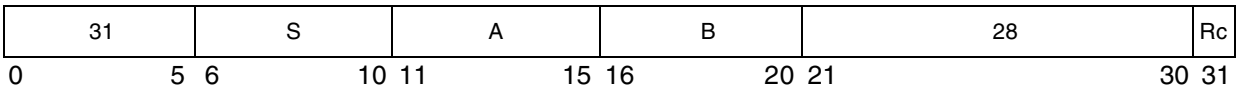
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

and*x* and*x*

AND

and **r**A,**r**S,**r**B (Rc = 0)
and. **r**A,**r**S,**r**B (Rc = 1)




```
rA ← (rS) & (rB)
```

The contents of **r**S are ANDed with the contents of **r**B and the result is placed into **r**A.

Other registers altered:

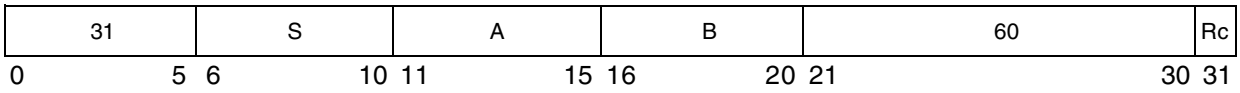
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

andc*x* andc*x*

AND with Complement

andc **r**A,**r**S,**r**B (Rc = 0)
andc. **r**A,**r**S,**r**B (Rc = 1)




```
rA ← (rS) + ¬ (rB)
```

The contents of **r**S are ANDed with the one's complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered:

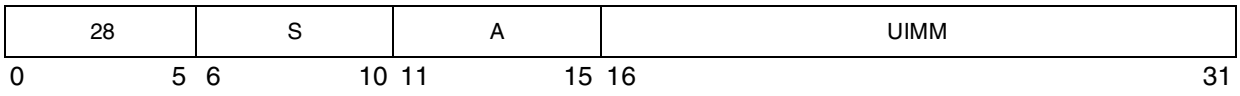
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

andi. andi.

AND Immediate

andi. **r**A,**r**S,UIMM




```
rA ← (rS) & ((16)0 || UIMM)
```

The contents of **r**S are ANDed with 0x0000 || UIMM and the result is placed into **r**A.

Other registers altered:

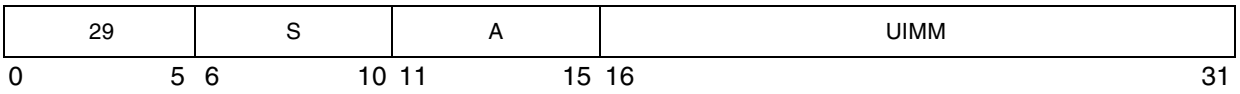
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO

Architecture Level	Supervisor Level	Optional	Form
UISA			D

andis. andis.

AND Immediate Shifted

andis. **r**A,**r**S,UIMM




```
rA ← (rS) + ( UIMM || (16)0)
```

The contents of **r**S are ANDed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

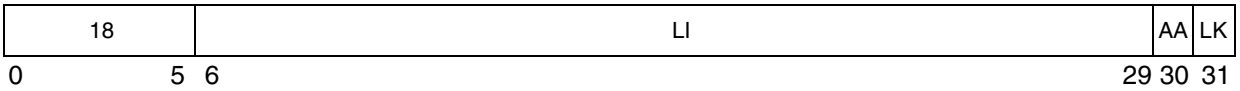
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO

Architecture Level	Supervisor Level	Optional	Form
UISA			D

b*x* b*x*

Branch

b	target_addr	(AA = 0 LK = 0)
ba	target_addr	(AA = 1 LK = 0)
bl	target_addr	(AA = 0 LK = 1)
bla	target_addr	(AA = 1 LK = 1)




```
if AA then NIA ←iea EXTS(LI || 0b00)
else NIA ←iea CIA + EXTS(LI || 0b00)
if LK then LR ←iea CIA + 4
```

target_addr specifies the branch target address.

If AA = 0, then the branch target address is the sum of LI || 0b00 sign-extended and the address of this instruction.

If AA = 1, then the branch target address is the value LI || 0b00 sign-extended.

If LK = 1, then the effective address of the instruction following the branch instruction is placed into the link register.

Other registers altered:

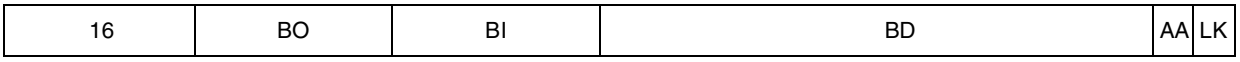
Affected: Link register (LR) (if LK = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			I

bc*x* bc*x*

Branch Conditional

bc BO,BI,target_addr (AA = 0 LK = 0)
bca BO,BI,target_addr (AA = 1 LK = 0)
bcl BO,BI,target_addr (AA = 0 LK = 1)
bcla BO,BI,target_addr (AA = 1 LK = 1)




```
if LK then LR ←iea CIA + 4
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
     if AA then NIA ←iea EXTS(BD || 0b00)
     else NIA ←iea CIA + EXTS(BD || 0b00)
```

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 8-6.

Table 8-6. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	0	00000	Negative (LT)—Set when the result is negative.
CR0[1]	1	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	2	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	3	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	4	00100	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	5	00101	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	6	00110	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	7	00111	Copy of FPSCR[OX] at the instruction's completion.
CR*n*[0]	8 12 16 20 24 28	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A < **fr**B.
CR*n*[1]	9 13 17 21 25 29	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions: **fr**A > **fr**B.
CR*n*[2]	10 14 18 22 26 30	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	11 15 19 23 27 31	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

Table 8-7 shows BO encodings. See also Section 4.2.4.2, “Conditional Branch Control.”

Table 8-7. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

Note: *z* bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture.
y bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

target_addr specifies the branch target address.

If AA = 0, the branch target address is the sum of BD || 0b00 sign-extended and the address of this instruction.

If AA = 1, the branch target address is the value BD || 0b00 sign-extended.

If LK = 1, the EA of the instruction following the branch instruction is placed into the LR.

Other registers altered:

Affected: Count register (CTR) (if BO[2] = 0)
Affected: Link register (LR) (if LK = 1)

Simplified mnemonics:

blt	target	equivalent to	**bc**	**12,0,**target
bne	**cr2**,target	equivalent to	**bc**	**4,10,**target
bdnz	target	equivalent to	**bc**	**16,0,**target

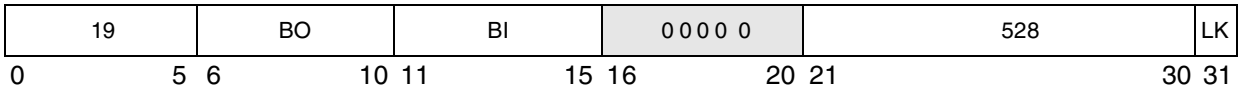
Architecture Level	Supervisor Level	Optional	Form
UISA			B

bcctr*x* bcctr*x*

Branch Conditional to Count Register

bcctr BO,BI (LK = 0)
bcctrl BO,BI (LK = 1)

Reserved




```
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if cond_ok then
 NIA ←iea CTR[0:29] || 0b00
 if LK then LR ←iea CIA + 4
```

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 8-8.

Table 8-8. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	0	00000	Negative (LT)—Set when the result is negative.
CR0[1]	1	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	2	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	3	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	4	00100	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	5	00101	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	6	00110	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	7	00111	Copy of FPSCR[OX] at the instruction's completion.
CR*n*[0]	8 12 16 20 24 28	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A < **fr**B.
CR*n*[1]	9 13 17 21 25 29	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A > **fr**B.
CR*n*[2]	10 14 18 22 26 30	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	11 15 19 23 27 31	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

Table 8-7 shows BO encodings. See also Section 4.2.4.2, “Conditional Branch Control.”

Table 8-9. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

Note: *z* bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture. *y* bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The branch target address is CTR[0–29] || 0b00.

If LK = 1, the EA of the instruction following the branch instruction is placed into the LR.

If “decrement and test CTR” is specified (BO[2] = 0), the instruction form is invalid.

Other registers altered:

Affected: Link register (LR) (if LK = 1)

Simplified mnemonics:

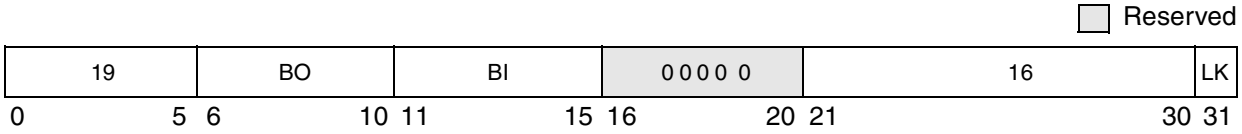
bltctr	equivalent to	**bcctr**	**12,0**
bnectr cr2	equivalent to	**bcctr**	**4,10**

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

bclr*x* bclr*x*

Branch Conditional to Link Register

bclr	BO,BI	(LK = 0)
bclrl	BO,BI	(LK = 1)




```
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
 NIA ←iea LR[0:29] || 0b00
 if LK then LR ←iea CIA + 4
```

The BI field specifies the CR bit used as the condition of the branch, as shown in Table 8-10.

Table 8-10. BI Operand Settings for CR Fields

CR*n* Bits	CR Bits	BI	Description
CR0[0]	0	00000	Negative (LT)—Set when the result is negative.
CR0[1]	1	00001	Positive (GT)—Set when the result is positive (and not zero).
CR0[2]	2	00010	Zero (EQ)—Set when the result is zero.
CR0[3]	3	00011	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.
CR1[0]	4	00100	Copy of FPSCR[FX] at the instruction's completion.
CR1[1]	5	00101	Copy of FPSCR[FEX] at the instruction's completion.
CR1[2]	6	00110	Copy of FPSCR[VX] at the instruction's completion.
CR1[3]	7	00111	Copy of FPSCR[OX] at the instruction's completion.
CR*n*[0]	8 12 16 20 24 28	01000 01100 10000 10100 11000 11100	Less than or floating-point less than (LT, FL). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A < **fr**B.
CR*n*[1]	9 13 17 21 25 29	01001 01101 10001 10101 11001 11101	Greater than or floating-point greater than (GT, FG). For integer compare instructions: **r**A > SIMM or **r**B (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison). For floating-point compare instructions:**fr**A > **fr**B.
CR*n*[2]	10 14 18 22 26 30	01010 01110 10010 10110 11010 11110	Equal or floating-point equal (EQ, FE). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B. For floating-point compare instructions: **fr**A = **fr**B.
CR*n*[3]	11 15 19 23 27 31	01011 01111 10011 10111 11011 11111	Summary overflow or floating-point unordered (SO, FU). For integer compare instructions, this is a copy of XER[SO] at the completion of the instruction. For floating-point compare instructions, one or both of **fr**A and **fr**B is a NaN.

Table 8-7 shows BO encodings. See also Section 4.2.4.2, “Conditional Branch Control.”

Table 8-11. BO Operand Encodings

BO	Description
0000*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.
0001*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.
001*zy*	Branch if the condition is FALSE.
0100*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.
0101*y*	Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.
011*zy*	Branch if the condition is TRUE.
1*z*00*y*	Decrement the CTR, then branch if the decremented CTR ≠ 0.
1*z*01*y*	Decrement the CTR, then branch if the decremented CTR = 0.
1*z*1*zz*	Branch always.

Note: *z* bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture. *y* bits provides a hint about whether a conditional branch is likely to be taken and may be used by some implementations to improve performance.

The branch target address is LR[0–29] || 0b00.

If LK = 1, the EA of the instruction following the branch instruction is placed into the LR.

Other registers altered:

Affected: Count register (CTR) (if BO[2] = 0)
Affected: Link register (LR) (if LK = 1)

Simplified mnemonics:

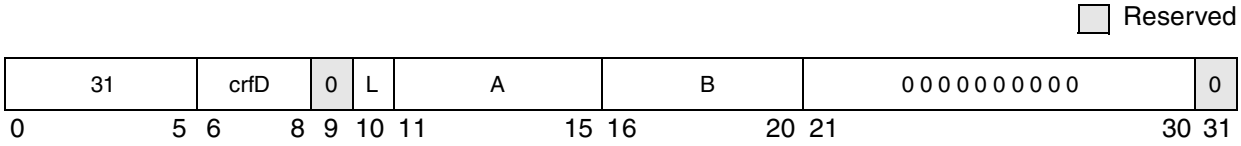
bltlr	equivalent to	**bclr 12,0**
bnelr cr2	equivalent to	**bclr 4,10**
bdnzlr	equivalent to	**bclr 16,0**

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

cmp cmp

Compare

cmp **crf**D,L,**r**A,**r**B




```
if L = 0 then a ← EXTS(rA)
         b ← EXTS(rB)
else     a ← (rA)
         b ← (rB)
if   a < b then c ← 0b100
else if a > b then c ← 0b010
else      c ← 0b001
CR[(4 * crfD):(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with the contents of **r**B, treating the operands as signed integers. The result of the comparison is placed into CR field **crf**D. The L bit has no effect on 32-bit operations.

Other registers altered:

- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

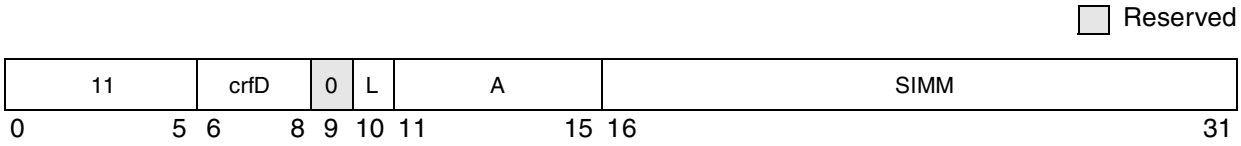
cmpw **crf**D,**r**A,**r**B equivalent to **cmp** **crf**D,**0**,**r**A,**r**B

Architecture Level	Supervisor Level	Optional	Form
UISA			X

cmpi cmpi

Compare Immediate

cmpi **crf**D,L,**r**A,SIMM




```
a ← (rA)
if   a < EXTS(SIMM) then c ← 0b100
else if a > EXTS(SIMM) then c ← 0b010
else      c ← 0b001
CR[(4 * crfD):(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with the sign-extended value of the SIMM field, treating the operands as signed integers. The result of the comparison is placed into CR field **crf**D.

In 32-bit implementations, if L = 1 the instruction form is invalid.

Other registers altered:

- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

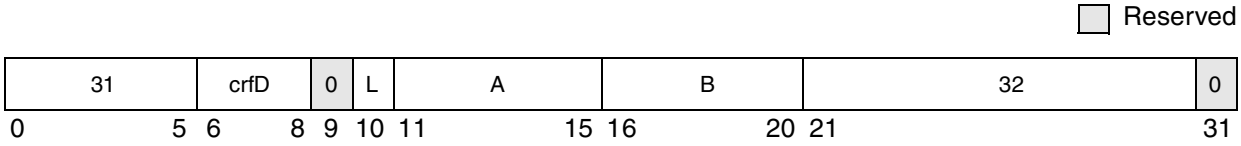
cmpwi **crf**D,**r**A,value equivalent to **cmpi** **crf**D,**0**,**r**A,value

Architecture Level	Supervisor Level	Optional	Form
UISA			D

cmpl cmpl

Compare Logical

cmpl **crf**D,L,**r**A,**r**B




```
a ← (rA)
b ← (rB)
if   a <U b then c ← 0b100
else if a >U b then c ← 0b010
else      c ← 0b001
CR[(4 * crfD):(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with the contents of **r**B, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

In 32-bit implementations, if L = 1 the instruction form is invalid.

Other registers altered:

- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

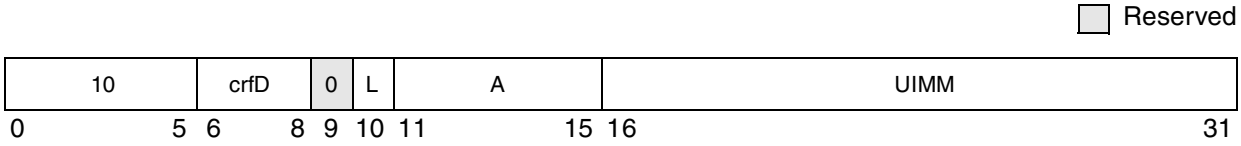
cmplw **crf**D,**r**A,**r**B equivalent to **cmpl** **crf**D,**0**,**r**A,**r**B

Architecture Level	Supervisor Level	Optional	Form
UISA			X

cmpli cmpli

Compare Logical Immediate

cmpli **crf**D,L,**r**A,UIMM




```
a ← (rA)
if   a <U ((16)0 || UIMM) then c ← 0b100
else if a >U ((16)0 || UIMM) then c ← 0b010
else     c ← 0b001
CR[(4 * crfD)–(4 * crfD + 3)] ← c || XER[SO]
```

The contents of **r**A are compared with 0x0000 || UIMM, treating the operands as unsigned integers. The result of the comparison is placed into CR field **crf**D.

In 32-bit implementations, if L = 1 the instruction form is invalid.

Other registers altered:

- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Simplified mnemonics:

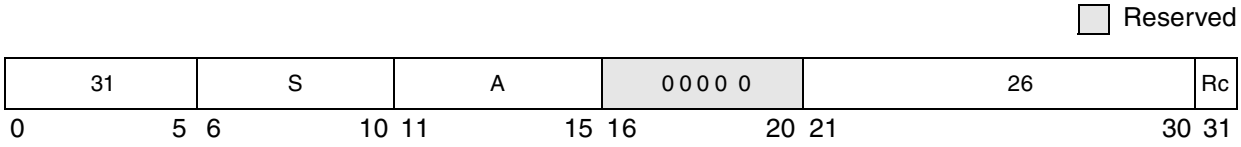
cmplwi **crf**D,**r**A,value equivalent to **cmpli** **crf**D,**0**,**r**A,value

Architecture Level	Supervisor Level	Optional	Form
UISA			D

cntlzw*x* cntlzw*x*

Count Leading Zeros Word

cntlzw **r**A,**r**S (Rc = 0)
cntlzw. **r**A,**r**S (Rc = 1)




```
n ← 0
do while n < 32
if rS[n] = 1 then leave
n ← n + 1
rA ← n
```

A count of the number of consecutive zero bits starting at bit 0 of **r**S is placed into **r**A. This number ranges from 0 to 32, inclusive.

Other registers altered:

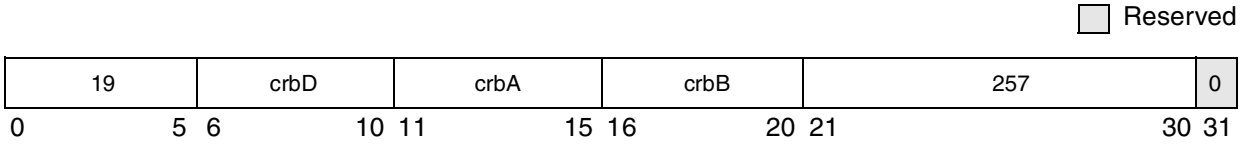
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: If Rc = 1, then LT is cleared in the CR0 field.

Architecture Level	Supervisor Level	Optional	Form
UISA			X

crand crand

Condition Register AND

crand **crb**D,**crb**A,**crb**B




```
CR[crbD] ← CR[crbA] & CR[crbB]
```

The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B. The result is placed into the CR bit specified by **crb**D.

Other registers altered:

- Condition register:

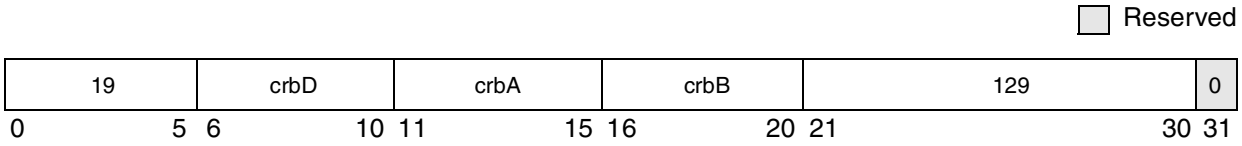
  Affected: Bit specified by operand **crb**D

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

crandc crandc

Condition Register AND with Complement

crandc **crb**D,**crb**A,**crb**B




```
CR[crbD] ← CR[crbA] & ¬ CR[crbB]
```

The CR bit specified by **crb**A is ANDed with the complement of the CR bit specified by **crb**B and the result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition register:
  Affected: Bit specified by operand **crb**D

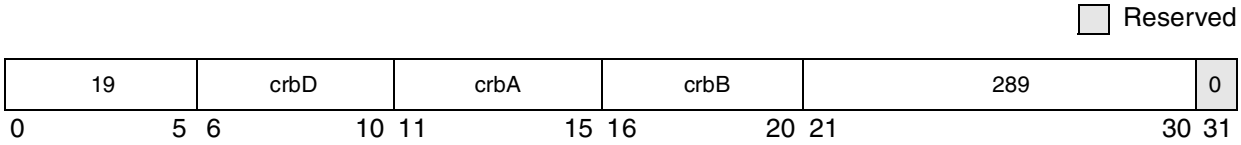
Architecture Level	Supervisor Level	Optional	Form
UISA			XL

creqv

creqv

Condition Register Equivalent

creqv **crb**D,**crb**A,**crb**B




```
CR[crbD] ← CR[crbA] ≡ CR[crbB]
```

The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition register:
  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

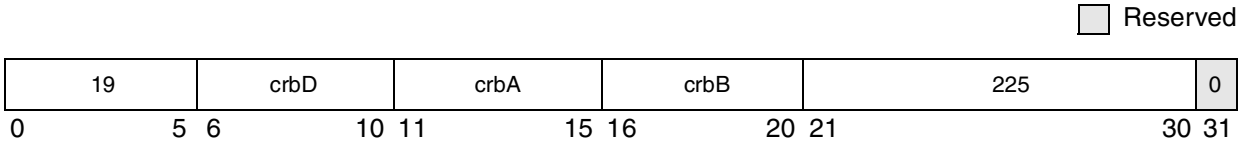
crset **crb**D equivalent to **creqv** **crb**D,**crb**D,**crb**D

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

crnand crnand

Condition Register NAND

crnand **crb**D,**crb**A,**crb**B




```
CR[crbD] ← ¬ (CR[crbA] & CR[crbB])
```

The CR bit specified by **crb**A is ANDed with the CR bit specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

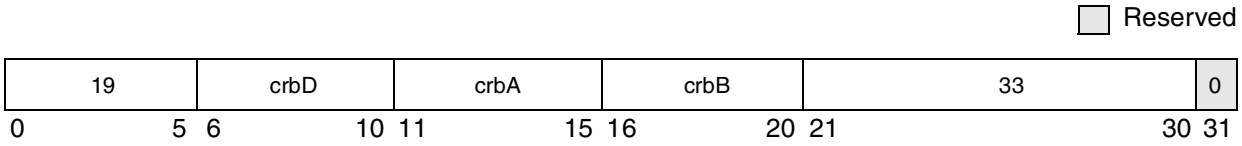
- Condition register:
  Affected: Bit specified by operand **crb**D

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

crnor crnor

Condition Register NOR

crnor **crb**D,**crb**A,**crb**B




```
CR[crbD] ← ¬ (CR[crbA] | CR[crbB])
```

The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B and the complemented result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition register:
  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

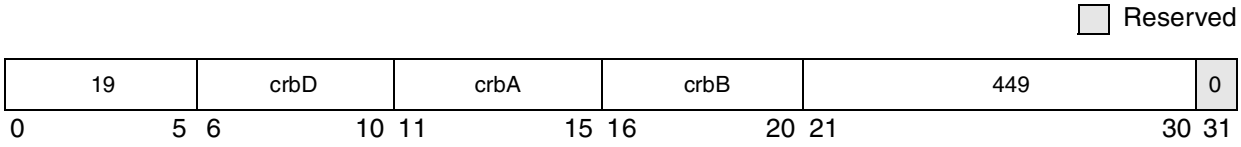
crnot **crb**D,**crb**A equivalent to **crnorcrb**D,**crb**A,**crb**A

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

cror cror

Condition Register OR

cror **crb**D,**crb**A,**crb**B




```
CR[crbD] ← CR[crbA] | CR[crbB]
```

The CR bit specified by **crb**A is ORed with the CR bit specified by **crb**B. The result is placed into the condition register bit specified by **crb**D.

Other registers altered:

- Condition register:
  Affected: Bit specified by operand **crb**D

Simplified mnemonics:

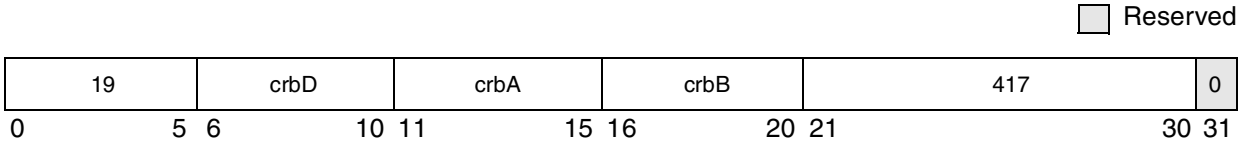
crmove crbD,**crb**A equivalent to **cror crb**D,**crb**A,**crb**A

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

crorc crorc

Condition Register OR with Complement

crorc **crb**D,**crb**A,**crb**B




```
CR[crbD] ← CR[crbA] | ¬ CR[crbB]
```

The CR bit specified by **crb**A is ORed with the complement of the condition register bit specified by **crb**B and the result is placed into the condition register bit specified by **crb**D.

Other registers altered:

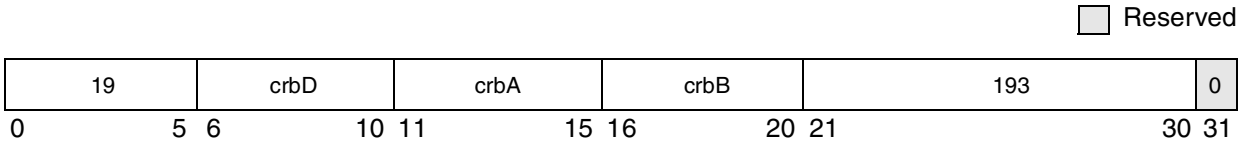
- Condition register:
  Affected: Bit specified by operand **crb**D

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

crxor crxor

Condition Register XOR

crxor **crb**D,**crb**A,**crb**B




```
CR[crbD] ← CR[crbA] ⊕ CR[crbB]
```

The CR bit specified by **crb**A is XORed with the CR bit specified by **crb**B and the result is placed into the CR bit specified by **crb**D.

Other registers altered:

- Condition register:
  Affected: Bit specified by **crb**D

Simplified mnemonics:

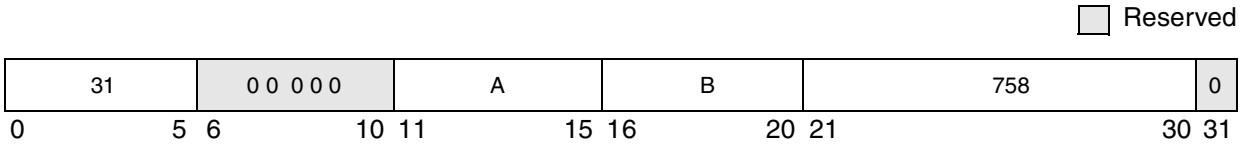
crclr **crb**D equivalent to **crxor** **crb**D,**crb**D,**crb**D

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

dcba dcba

Data Cache Block Allocate

dcba **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

dcba allocates the data cache block addressed by EA by marking it valid without reading the contents of the block from memory; data in the cache block is considered undefined after **dcba** completes. **dcba** is a hint that the program will probably soon store into a portion of the block, but the contents of the rest of the block are not meaningful to the program (eliminating the need to read the entire block from main memory), and can provide for improved performance in these code sequences. **dcba** executes as follows:

- If the cache block containing the byte addressed by EA is in the data cache, the contents of all bytes are made undefined but the cache block is still considered valid. Note that programming errors can occur if the data in this cache block is subsequently read or used inadvertently.
- If the cache block containing the byte addressed by EA is not in the data cache and the corresponding space is caching-allowed, the cache block is allocated and made valid without fetching the block from main memory. All byte values are undefined.
- If the addressed byte corresponds to a caching-inhibited page or block (the I bit is set), **dcba** is treated as a no-op.
- If the cache block containing the byte addressed by EA is coherency-required and it exists in the any other processor's cache, those processors perform the appropriate bus transactions to enforce coherency.

dcba acts as a store to the addressed byte with respect to translation, reference and change recording, memory protection, and ordering enforced by **eieio** or by a combination of caching-inhibited and guarded attributes. However, a DSI interrupt is not invoked for a translation or protection violation and the reference and change bits need not be updated when the page or block is cache-inhibited (causing **dcba** to be treated as a no-op).

Other registers altered: None

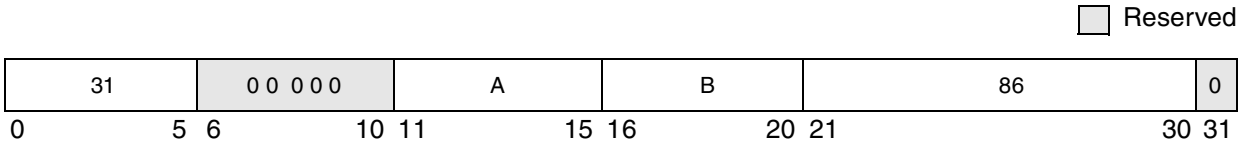
The OEA defines **dcba** to clear all bytes of a new cache block it was not in the cache. Additionally, as **dcba** may establish a data cache block without verifying that the associated physical address is valid, a delayed machine check interrupt may occur.

Architecture Level	Supervisor Level	Optional	Form
VEA		√	X

dcbf dcbf

Data Cache Block Flush

dcbf **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

dcbf invalidates the block in the data cache addressed by EA, copying the block to memory first if it contains modified data. If the block is marked coherency-required in a multiprocessor system, if necessary the processor sends an address-only broadcast to other processors. If the block is modified in another processor, broadcasting **dcbf** causes the processor to copy the block to memory and invalidate the block. The action taken depends on the memory mode associated with the block containing the byte addressed by EA and on the state of that block. The list below describes the action taken for the various states of the memory coherency attribute (M bit).

- Coherency required
  - Unmodified block—Invalidates copies of the data cache block in all processors.
  - Modified block—Copies the block to memory. Invalidates copies of the block in the data caches of all processors.
  - Absent block—If modified copies of the block are in the data caches of other processors, causes them to be copied to memory and invalidated in those data caches. If unmodified copies are in the data caches of other processors, causes those copies to be invalidated in those data caches.
- Coherency not required
  - Unmodified block—Invalidates the block in the processor's data cache.
  - Modified block—Copies the block to memory. Invalidates the block in the processor's data cache.
  - Absent block (target block not in cache)—No action is taken.

The function of **dcbf** is independent of the write-through, write-back and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for reference and change bit recording except that reference and change bit recording may not occur.

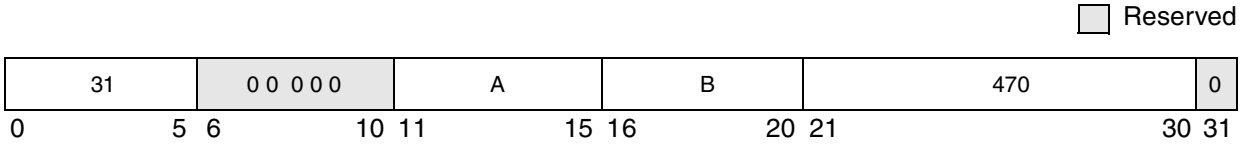
Other registers altered: None

Architecture Level	Supervisor Level	Optional	Form
VEA			X

dcbi dcbi

Data Cache Block Invalidate

dcbi **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

The action taken is dependent on the memory mode associated with the block containing the byte addressed by EA and on the state of that block. The list below describes the action taken if the block containing the byte addressed by EA is or is not in the cache.

- Coherency required
  - — Unmodified block—Invalidates copies of the block in the data caches of all processors.
  - — Modified block—Invalidates copies of the block in the data caches of all processors. (Discards the modified contents.)
  - — Absent block—If copies of the block are in the data caches of any other processor, causes the copies to be invalidated in those data caches. (Discards any modified contents.)
- Coherency not required
  - — Unmodified block—Invalidates the block in the processor's data cache.
  - — Modified block—Invalidates the block in the processor's data cache. (Discards the modified contents.)
  - — Absent block (target block not in cache)—No action is taken.

When data address translation is enabled, MSR[DR] = 1 and the virtual address has no translation, a DSI interrupt occurs.

The function of **dcbi** is independent of the write-through and caching-inhibited/allowed modes of the block containing the byte addressed by EA. This instruction operates as a store to the addressed byte with respect to address translation and protection. The reference and change bits are modified appropriately.

Note that some implementations execute this instruction as a **dcbf**.

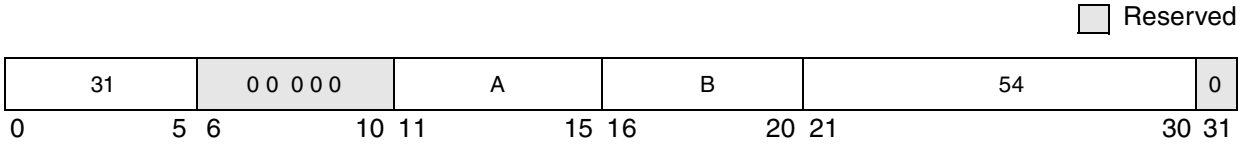
Other registers altered: None

Architecture Level	Supervisor Level	Optional	Form
OEA	√		X

dcbst dcbst

Data Cache Block Store

dcbst **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

The **dcbst** instruction executes as follows:

- If the block containing the byte addressed by EA is in coherency-required mode, and a block containing the byte addressed by EA is in the data cache of any processor and has been modified, the writing of it to main memory is initiated.
- If the block containing the byte addressed by EA is in coherency-not-required mode, and a block containing the byte addressed by EA is in the data cache of this processor and has been modified, the writing of it to main memory is initiated.

The function of this instruction is independent of the write-through and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

The coherency state of a cache block after a write access (caused by a **dcbst**) is implementation-dependent. For example, some implementations may mark the cache block exclusive, where others may mark it invalid.

The processor treats this instruction as a load from the addressed byte with respect to address translation and memory protection. It is also treated as a load for reference and change bit recording except that reference and change bit recording may not occur.

Other registers altered:

- None

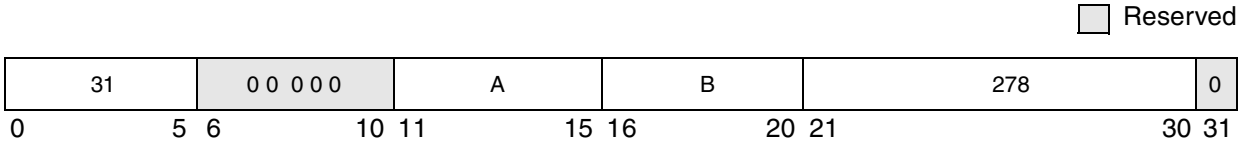
Architecture Level	Supervisor Level	Optional	Form
VEA			X

dcbt

dcbt

Data Cache Block Touch

dcbt **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

This instruction is a hint that performance will possibly be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon load from the addressed byte. If the block is caching-inhibited, the hint is ignored and the instruction is treated as a no-op. Executing **dcbt** does not cause the system alignment error handler to be invoked.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording except that reference and change bit recording may not occur. Additionally, no interrupt occurs in the case of a translation fault or protection violation.

The program uses the **dcbt** instruction to request a cache block fetch before it is actually needed by the program. The program can later execute load instructions to put data into registers. However, the processor is not obliged to load the addressed block into the data cache. Note that this instruction is defined architecturally to perform the same functions as the **dcbtst** instruction. Both are defined to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

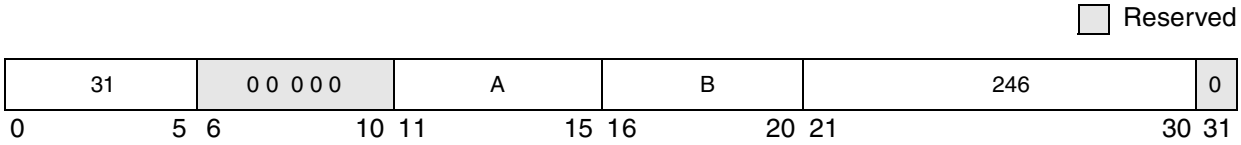
- None

Architecture Level	Supervisor Level	Optional	Form
VEA			X

dcbtst dcbtst

Data Cache Block Touch for Store

dcbtst **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

This instruction is a hint that performance will possibly be improved if the block containing the byte addressed by EA is fetched into the data cache, because the program will probably soon store from the addressed byte. If the block is caching-inhibited, the hint is ignored and the instruction is treated as a no-op. Executing **dcbtst** does not cause the system alignment error handler to be invoked.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, and reference and change recording except that reference and change bit recording may not occur. Additionally, no interrupt occurs in the case of a translation fault or protection violation.

The program uses **dcbtst** to request a cache block fetch to potentially improve performance for a subsequent store to that EA, as that store would then be to a cached location. However, the processor is not obliged to load the addressed block into the data cache. Note that this instruction is defined architecturally to perform the same functions as the **dcbt** instruction. Both are defined to allow implementations to differentiate the bus actions when fetching into the cache for the case of a load and for a store.

Other registers altered:

- None

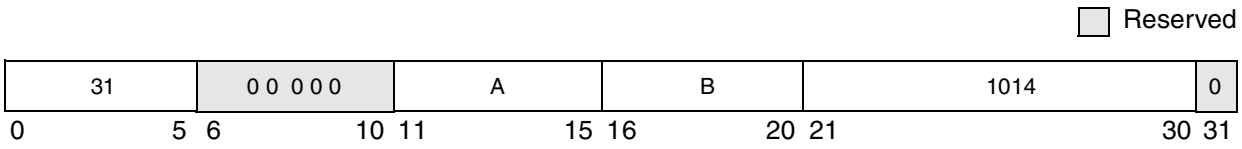
Architecture Level	Supervisor Level	Optional	Form
VEA			X

dcbz

dcbz

Data Cache Block Clear to Zero

dcbz **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

The **dcbz** instruction executes as follows:

- If the cache block containing the byte addressed by EA is in the data cache, all bytes are cleared.
- If the cache block containing the byte addressed by EA is not in the data cache and the corresponding memory page or block is caching-allowed, the cache block is allocated (and made valid) in the data cache without fetching the block from main memory, and all bytes are cleared.
- If the page containing the byte addressed by EA is in caching-inhibited or write-through mode, either all bytes of main memory that correspond to the addressed cache block are cleared or the alignment interrupt handler is invoked. The interrupt handler can then clear all bytes in main memory that correspond to the addressed cache block.
- If the cache block containing the byte addressed by EA is in coherency-required mode, and the cache block exists in the data cache(s) of any other processor(s), it is kept coherent in those caches (i.e. the processor performs the appropriate bus transactions to enforce this).

This instruction is treated as a store to the addressed byte with respect to address translation, memory protection, reference and change recording. It is also treated as a store with respect to the ordering enforced by **eieio** and the ordering enforced by the combination of caching-inhibited and guarded attributes for a page (or block).

Other registers altered:

- None

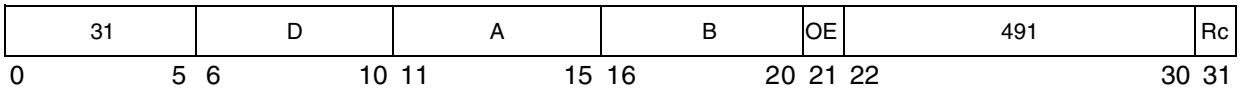
The OEA describes how **dcbz** may establish a data cache block without verifying that the associated physical address is valid. This can cause a delayed machine check interrupt; see Chapter 6, “Interrupts.”

Architecture Level	Supervisor Level	Optional	Form
VEA			X

divw*x* divw*x*

Divide Word

divw	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
divw.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
divwo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
divwo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
dividend ← (rA)
divisor ← (rB)
rD ← dividend ÷ divisor
```

The dividend is the contents of **r**A. The divisor is the contents of **r**B. The 32-bit quotient is formed and placed in **r**D. The remainder is not supplied as a result.

Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation—dividend = (quotient * divisor) + r where $0 \le r < |\text{divisor}|$ (if the dividend is non-negative), and $-|\text{divisor}| < r \le 0$ (if the dividend is negative).

If an attempt is made to perform either of the divisions—0x8000_0000 ÷ –1 or <anything> ÷ 0, then the contents of **r**D are undefined, as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = 1). In this case, if OE = 1 then OV is set.

The 32-bit signed remainder of dividing the contents of **r**A by the contents of **r**B can be computed as follows, except in the case that the contents of **r**A = -2^{31} and the contents of **r**B = –1.

divw	**r**D,**r**A,**r**B	# **r**D = quotient
mullw	**r**D,**r**D,**r**B	# **r**D = quotient ∗ divisor
subf	**r**D,**r**D,**r**A	# **r**D = remainder

Other registers altered:

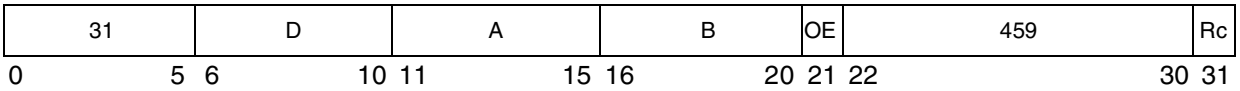
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: SO, OV (if OE = 1)
  Note: The setting of the affected XER bits is mode-independent and reflects overflow of the 32-bit result.

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

divwu*x* divwu*x*

Divide Word Unsigned

divwu	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
divwu.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
divwuo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
divwuo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
dividend ← (rA)
divisor ← (rB)
rD ← dividend ÷ divisor
```

The dividend is the contents of **r**A. The divisor is the contents of **r**B. A 32-bit quotient is formed. The 32-bit quotient is placed into **r**D. The remainder is not supplied as a result.

Both operands and the quotient are interpreted as unsigned integers, except that if Rc = 1 the first three bits of CR0 field are set by signed comparison of the result to zero. The quotient is the unique unsigned integer that satisfies the equation—dividend = (quotient ∗ divisor) + r (where 0 ≤ r < divisor). If an attempt is made to perform the division—<anything> ÷ 0—then the contents of **r**D are undefined as are the contents of the LT, GT, and EQ bits of the CR0 field (if Rc = 1). In this case, if OE = 1 then OV is set.

The 32-bit unsigned remainder of dividing the contents of **r**A by the contents of **r**B can be computed as follows:

divwu	**r**D,**r**A,**r**B	# **r**D = quotient
mullw	**r**D,**r**D,**r**B	# **r**D = quotient ∗ divisor
subf	**r**D,**r**D,**r**A	# **r**D = remainder

Other registers altered:

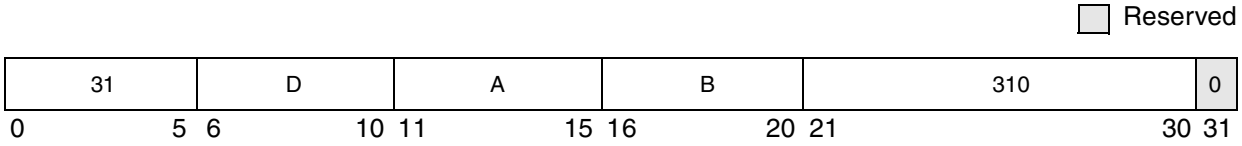
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: SO, OV (if OE = 1)
  Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 32-bit result.

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

eciwx eciwx

External Control In Word Indexed

eciwx **r**D,**r**A,**r**B



The **eciwx** instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```
if rA = 0 then b ← 0
else b← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send load word request for paddr to device identified by EAR[RID]
rD ← word from device
```

EA is the sum (**r**A|0) + (**r**B).

A load word request for the physical address (referred to as real address in the architecture specification) corresponding to EA is sent to the device identified by EAR[RID], bypassing the cache. The word returned by the device is placed in **r**D.

EAR[E] must be 1. If it is not, a DSI interrupt is generated.

EA must be a multiple of four. If it is not, one of the following occurs:

- A system alignment interrupt is generated.
- A DSI interrupt is generated (possible only if EAR[E] = 0).
- The results are boundedly undefined.

The **eciwx** instruction is supported for EAs that reference memory segments in which SR[T] = 1 and for EAs mapped by DBAT registers. If this instruction is executed when MSR[DR] = 0 (real addressing mode), the results are boundedly undefined.

This instruction is treated as a load from the addressed byte with respect to address translation, memory protection, reference and change bit recording, and the ordering performed by **eieio**.

This instruction is optional in the architecture.

Other registers altered:

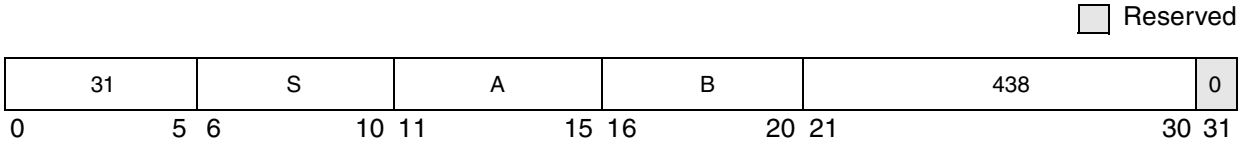
- None

Architecture Level	Supervisor Level	Optional	Form
VEA		√	X

ecowx ecowx

External Control Out Word Indexed

ecowx **r**S,**r**A,**r**B



The **ecowx** instruction and the EAR register can be very efficient when mapping special devices such as graphics devices that use addresses as pointers.

```
if rA = 0 then b ← 0
else b ← (rA)
EA ← b + (rB)
paddr ← address translation of EA
send store word request for paddr to device identified by EAR[RID]
send rS to device
```

EA is the sum (**r**A|0) + (**r**B).

A store word request for the physical address corresponding to EA and the contents of **r**S are sent to the device identified by EAR[RID], bypassing the cache.

If EAR[E] = 0, a DSI interrupt is generated. If EA is not a multiple of four, one of the following occurs:

- A system alignment interrupt is generated.
- A DSI interrupt is generated (possible only if EAR[E] = 0).
- The results are boundedly undefined.

The **ecowx** instruction is supported for EAs that reference memory segments in which SR[T] = 0), and for EAs mapped by DBATs. If this instruction is executed when MSR[DR] = 0 (real addressing mode), results are boundedly undefined.

ecowx is treated as a store from the addressed byte with respect to address translation, memory protection, reference and change bit recording, and the ordering performed by **eieio**. Software synchronization is required to ensure that the data access is performed in program order with respect to data accesses caused by other store or **ecowx** instructions, even though the addressed byte is assumed to be caching-inhibited and guarded.

This instruction is optional in the architecture.

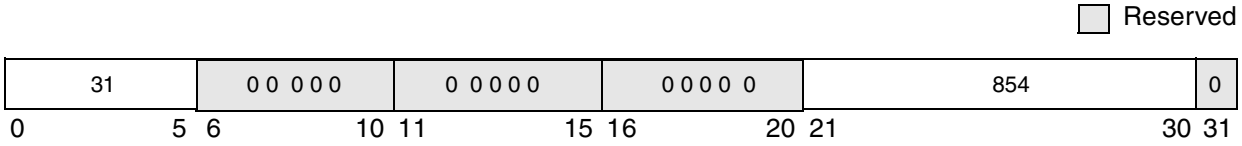
Other registers altered:

- None

Architecture Level	Supervisor Level	Optional	Form
VEA		√	X

eieio eieio

Enforce In-Order Execution of I/O



The **eieio** instruction provides an ordering function for the effects of load and store instructions executed by a processor. These loads and stores are divided into two sets, which are ordered separately. The memory accesses caused by a **dcbz** or a **dcba** instruction are ordered like a store. The two sets follow:

1. Loads and stores to memory that is both caching-inhibited and guarded, and stores to memory that is write-through required.

   The **eieio** instruction controls the order in which the accesses are performed in main memory. It ensures that all applicable memory accesses caused by instructions preceding the **eieio** instruction have completed with respect to main memory before any applicable memory accesses caused by instructions following the **eieio** instruction access main memory. It acts like a barrier that flows through the memory queues and to main memory, preventing the reordering of memory accesses across the barrier. No ordering is performed for **dcbz** if the instruction causes the system alignment error handler to be invoked.

   All accesses in this set are ordered as a single set—that is, there is not one order for loads and stores to caching-inhibited and guarded memory and another order for stores to write-through required memory.

2. Stores to memory that have all of the following attributes—caching-allowed, write-through not required, and memory-coherency required.

   The **eieio** instruction controls the order in which the accesses are performed with respect to coherent memory. It ensures that all applicable stores caused by instructions preceding the **eieio** instruction have completed with respect to coherent memory before any applicable stores caused by instructions following the **eieio** instruction complete with respect to coherent memory.

With the exception of **dcbz** and **dcba**, **eieio** does not affect the order of cache operations (whether caused explicitly by execution of a cache management instruction, or implicitly by the cache coherency mechanism). See Chapter 5, “Cache Model and Memory Coherency.” The **eieio** instruction does not affect the order of accesses in one set with respect to accesses in the other set.

The **eieio** instruction may complete before memory accesses caused by instructions preceding the **eieio** instruction have been performed with respect to main memory or coherent memory as appropriate.

The **eieio** instruction is intended for use in managing shared data structures, in accessing memory-mapped I/O, and in preventing load/store combining operations in main memory. For the first use, the shared data structure and the lock that protects it must be altered only by stores that are in the same set (1 or 2; see previous discussion). For the second use, **eieio** can be thought of as placing a barrier into the stream of memory accesses issued by a processor, such that any given memory access appears to be on the same side of the barrier to both the processor and the I/O device.

Because the processor performs store operations in order to memory that is designated as both caching-inhibited and guarded (refer to Section 5.2.1, “Memory Access Ordering”), **eieio** is needed for such memory only when loads must be ordered with respect to stores or with respect to other loads.

Note that **eieio** does not connect hardware considerations to it such as multiprocessor implementations that send an **eieio** address-only broadcast (useful in some designs). For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, **eieio** broadcasts signals to that buffer that previous loads/stores (marked caching-inhibited, guarded, or write-through required) must complete before any following loads/stores (marked caching-inhibited, guarded, or write-through required).

Other registers altered:

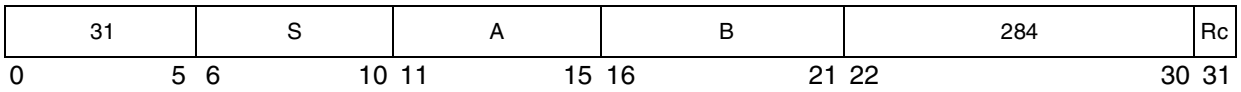
- None

Architecture Level	Supervisor Level	Optional	Form
VEA			X

eqv*x* eqv*x*

Equivalent

eqv **r**A,**r**S,**r**B (Rc = 0)
eqv. **r**A,**r**S,**r**B (Rc = 1)




```
rA ← (rS) ≡ (rB)
```

The contents of **r**S are XORed with the contents of **r**B and the complemented result is placed into **r**A.

Other registers altered:

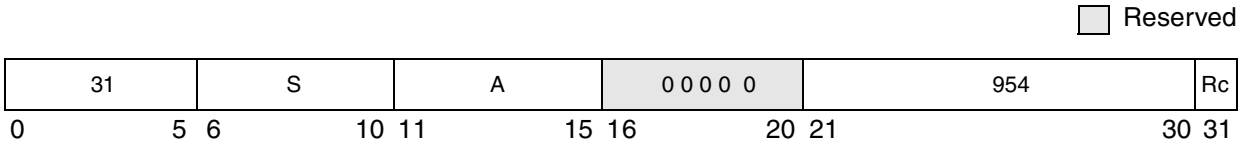
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

extsb*x* extsb*x*

Extend Sign Byte

extsb **r**A,**r**S (Rc = 0)
extsb. **r**A,**r**S (Rc = 1)




```
S ← rS[24]
rA[24:31] ← rS[24:31]
rA[0:23] ← (24)S
```

The contents of **r**S[24–31] are placed into **r**A[24–31]. Bit 24 of **r**S is placed into **r**A[0–23].

Other registers altered:

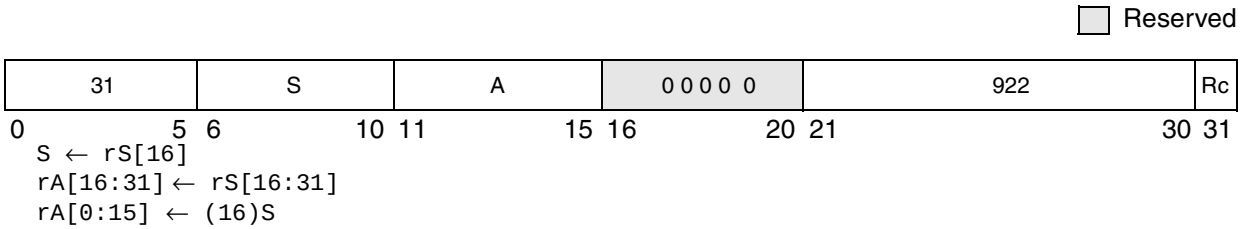
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

extsh*x* extsh*x*

Extend Sign Half Word

extsh **r**A,**r**S (Rc = 0)
extsh. **r**A,**r**S (Rc = 1)




```
S ← rS[16]
rA[16:31] ← rS[16:31]
rA[0:15] ← (16)S
```

The contents of **r**S[16–31] are placed into **r**A[16–31]. Bit 16 of **r**S is placed into **r**A[0–15].

Other registers altered:

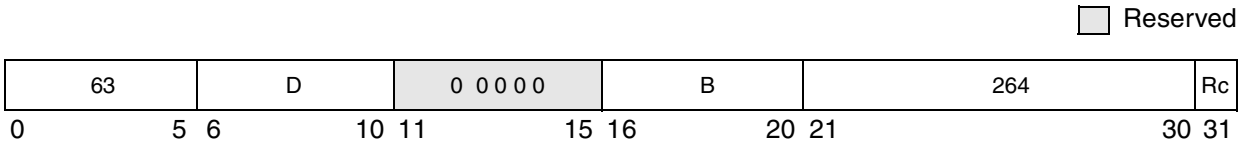
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fabs*x* fabs*x*

Floating Absolute Value

fabs **fr**D,**fr**B (Rc = 0)
fabs. **fr**D,**fr**B (Rc = 1)



The contents of **fr**B with bit 0 cleared are placed into **fr**D.

Note that the **fabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fabs**. This instruction does not alter the FPSCR.

Other registers altered:

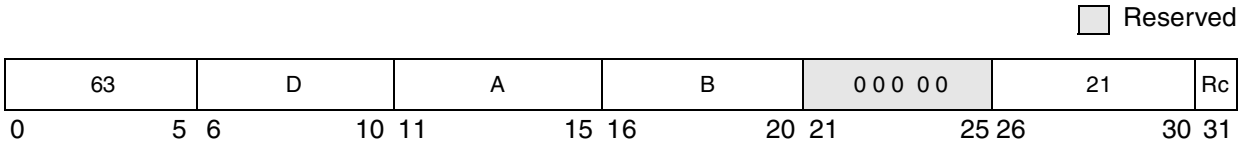
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fadd*x* fadd*x*

Floating Add (Double-Precision)

fadd **fr**D,**fr**A,**fr**B (Rc = 0)
fadd. **fr**D,**fr**A,**fr**B (Rc = 1)



The floating-point operand in **fr**A is added to the floating-point operand in **fr**B. If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

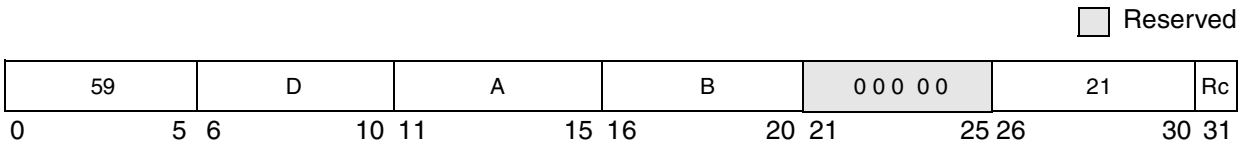
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX,VXSNAN, VXISI

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fadds*x* fadds*x*

Floating Add Single

fadds **fr**D,**fr**A,**fr**B (Rc = 0)
fadds. **fr**D,**fr**A,**fr**B (Rc = 1)



The floating-point operand in **fr**A is added to the floating-point operand in **fr**B. If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to the single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point addition is based on exponent comparison and addition of the two significands. The exponents of the two operands are compared, and the significand accompanying the smaller exponent is shifted right, with its exponent increased by one for each bit shifted, until the two exponents are equal. The two significands are then added or subtracted as appropriate, depending on the signs of the operands. All 53 bits in the significand as well as all three guard bits (G, R, and X) enter into the computation.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

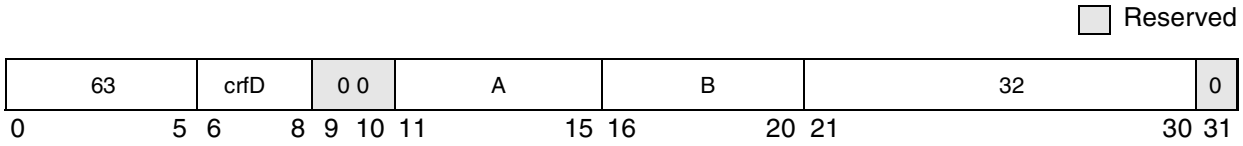
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX,VXSNAN, VXISI

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fcmpo fcmpo

Floating Compare Ordered

fcmpo **crf**D,**fr**A,**fr**B




```
if (frA) is a NaN or
   (frB) is a NaN then c ← 0b0001
else if (frA)< (frB) thenc ← 0b1000
else if (frA)> (frB) thenc ← 0b0100
else                       c ← 0b0010
FPCC ← c
CR[(4 * crfD):(4 * crfD + 3)] ← c
if (frA) is an SNaN or
   (frB) is an SNaN then
        VXSNAN ← 1
        if VE = 0 then VXVC ← 1
else if (frA) is a QNaN or
   (frB) is a QNaN then VXVC ← 1
```

The floating-point operand in **fr**A is compared to the floating-point operand in **fr**B. The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set, and if invalid operation is disabled (VE = 0) then VXVC is set. Otherwise, if one of the operands is a QNaN, then VXVC is set.

Other registers altered:

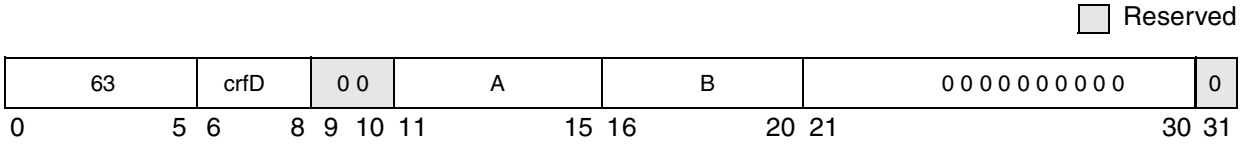
- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, UN
- Floating-point status and control register:
  Affected: FPCC, FX, VXSNAN, VXVC

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fcmpu fcmpu

Floating Compare Unordered

fcmpu **crf**D,**fr**A,**fr**B




```
if (frA) is a NaN or
    (frB) is a NaN then    c ← 0b0001
else if (frA) < (frB) then c ← 0b1000
else if (frA) > (frB) then c ← 0b0100
else                       c ← 0b0010
FPCC ← c
CR[(4 * crfD):(4 * crfD + 3)] ← c
if (frA) is an SNaN or
    (frB) is an SNaN then
         VXSNAN ← 1
```

The floating-point operand in register **fr**A is compared to the floating-point operand in register **fr**B. The result of the compare is placed into CR field **crf**D and the FPCC.

If one of the operands is a NaN, either quiet or signaling, then CR field **crf**D and the FPCC are set to reflect unordered. If one of the operands is a signaling NaN, then VXSNAN is set.

Other registers altered:

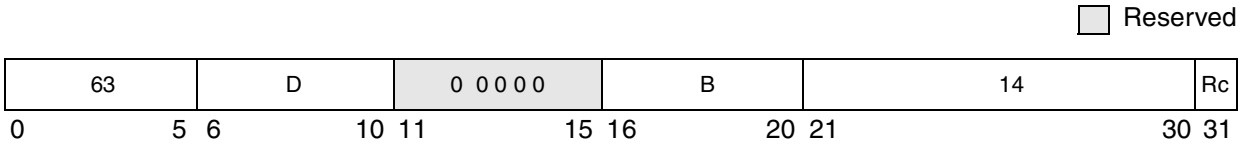
- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, UN
- Floating-point status and control register:
  Affected: FPCC, FX, VXSNAN

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fctiw*x* fctiw*x*

Floating Convert to Integer Word

fctiw **fr**D,**fr**B (Rc = 0)
fctiw. **fr**D,**fr**B (Rc = 1)



The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode specified by FPSCR[RN], and placed in bits 32–63 of **fr**D. Bits 0–31 of **fr**D are undefined.

If the operand in **fr**B are greater than $2^{31} - 1$, bits 32–63 of **fr**D are set to 0x7FFF_FFFF.

If the operand in **fr**B are less than -2^{31}, bits 32–63 of **fr**D are set to 0x8000_0000.

The conversion is described fully in Section C.4.2, “Floating-Point Convert to Integer Model.”

Except for trap-enabled invalid operation interrupts, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Other registers altered:

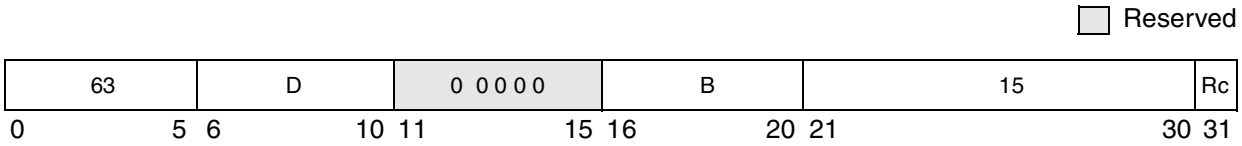
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fctiwz*x* fctiwz*x*

Floating Convert to Integer Word with Round toward Zero

fctiwz **fr**D,**fr**B (Rc = 0)
fctiwz. **fr**D,**fr**B (Rc = 1)



The floating-point operand in register **fr**B is converted to a 32-bit signed integer, using the rounding mode round toward zero, and placed in bits 32–63 of **fr**D. Bits 0–31 of **fr**D are undefined.

If the operand in **fr**B is greater than $2^{31} - 1$, bits 32–63 of **fr**D are set to 0x7FFF_FFFF.

If the operand in **fr**B is less than -2^{31}, bits 32–63 of **fr**D are set to 0x 8000_0000.

The conversion is described fully in Section C.4.2, “Floating-Point Convert to Integer Model.”

Except for trap-enabled invalid operation interrupts, FPSCR[FPRF] is undefined. FPSCR[FR] is set if the result is incremented when rounded. FPSCR[FI] is set if the result is inexact.

Other registers altered:

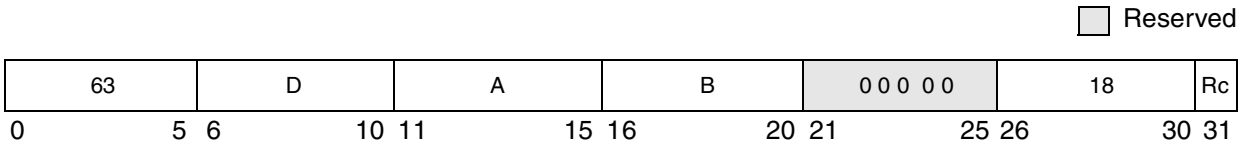
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF (undefined), FR, FI, FX, XX, VXSNAN, VXCVI

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fdivx fdivx

Floating Divide (Double-Precision)

fdiv **fr**D,**fr**A,**fr**B (Rc = 0)
fdiv. **fr**D,**fr**A,**fr**B (Rc = 1)



The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. The remainder is not supplied as a result.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1 and zero divide interrupts when FPSCR[ZE] = 1.

Other registers altered:

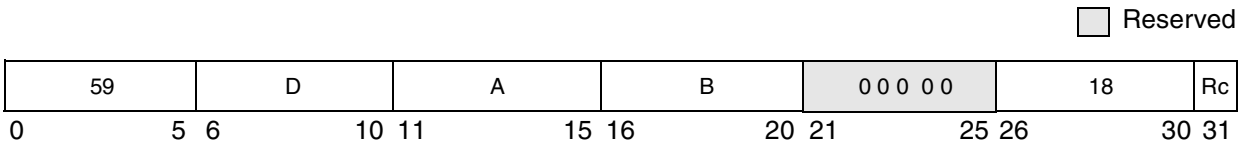
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fdivs*x* fdivs*x*

Floating Divide Single

fdivs **fr**D,**fr**A,**fr**B (Rc = 0)
fdivs. **fr**D,**fr**A,**fr**B (Rc = 1)



The floating-point operand in register **fr**A is divided by the floating-point operand in register **fr**B. The remainder is not supplied as a result.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point division is based on exponent subtraction and division of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1 and zero divide interrupts when FPSCR[ZE] = 1.

Other registers altered:

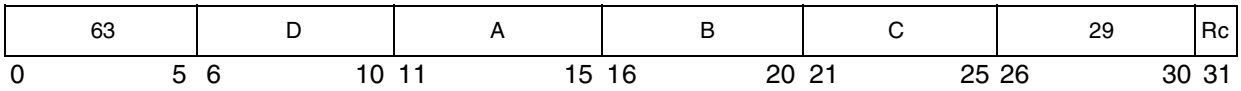
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, ZX, XX, VXSNAN, VXIDI, VXZDZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fmadd*x* fmadd*x*

Floating Multiply-Add (Double-Precision)

fmadd **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fmadd. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← (frA * frC) + frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

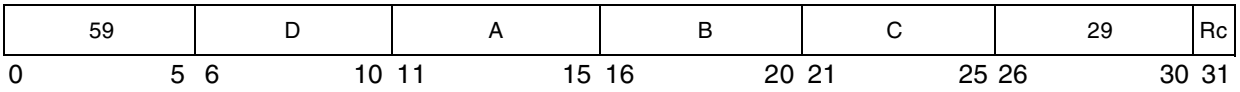
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fmadds*x* fmadds*x*

Floating Multiply-Add Single

fmadds **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fmadds. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← (frA * frC) + frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

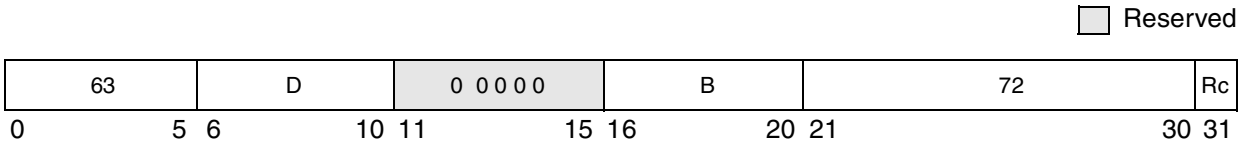
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fmr*x* fmr*x*

Floating Move Register

fmr **fr**D,**fr**B (Rc = 0)
fmr. **fr**D,**fr**B (Rc = 1)



The contents of register **fr**B are placed into **fr**D.

Other registers altered:

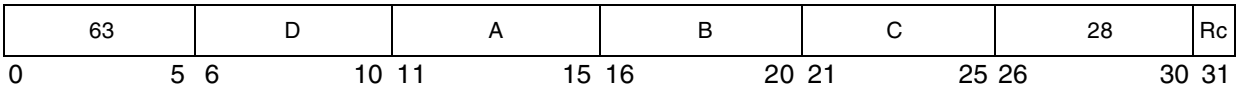
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fmsub*x* fmsub*x*

Floating Multiply-Subtract (Double-Precision)

fmsub **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fmsub. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← [frA * frC] – frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

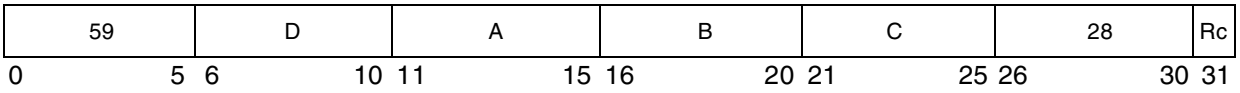
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fmsubs*x* fmsubs*x*

Floating Multiply-Subtract Single

fmsubs **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fmsubs. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← [frA * frC] – frB
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

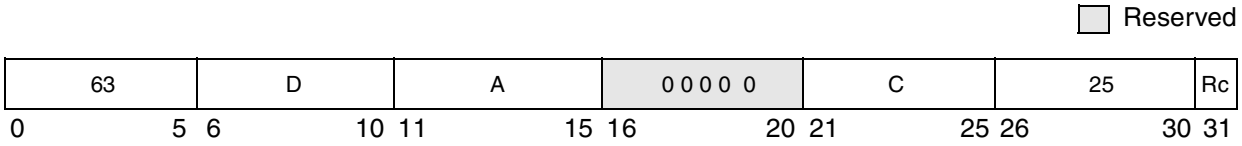
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fmul*x* fmul*x*

Floating Multiply (Double-Precision)

fmul **fr**D,**fr**A,**fr**C (Rc = 0)
fmul. **fr**D,**fr**A,**fr**C (Rc = 1)



The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

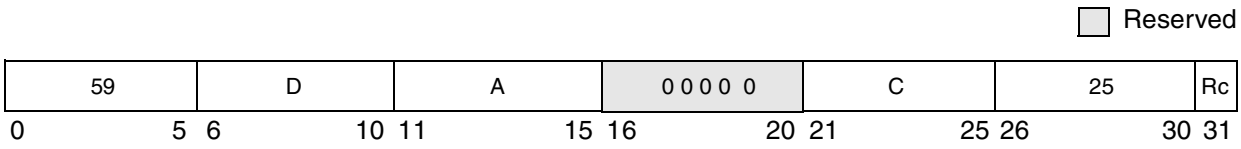
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fmuls*x* fmuls*x*

Floating Multiply Single

fmuls **fr**D,**fr**A,**fr**C (Rc = 0)
fmuls. **fr**D,**fr**A,**fr**C (Rc = 1)



The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C.

If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

Floating-point multiplication is based on exponent addition and multiplication of the significands.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation interrupts when FPSCR[VE] = 1.

Other registers altered:

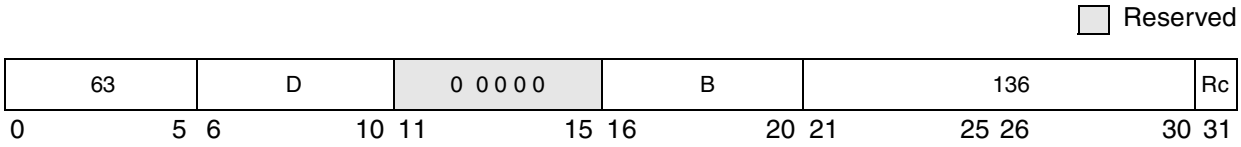
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fnabs*x* fnabs*x*

Floating Negative Absolute Value

fnabs **fr**D,**fr**B (Rc = 0)
fnabs. **fr**D,**fr**B (Rc = 1)



The contents of register **fr**B with bit 0 set are placed into **fr**D.

Note that the **fnabs** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fnabs**. This instruction does not alter the FPSCR.

Other registers altered:

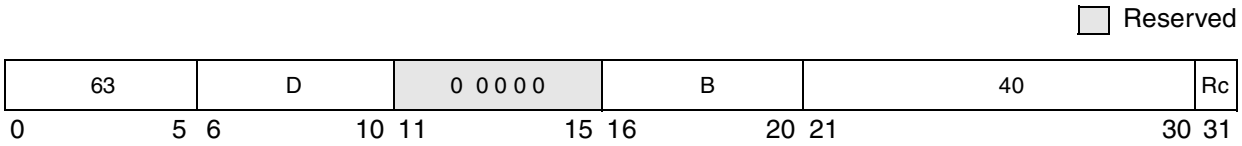
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fneg*x* fneg*x*

Floating Negate

fneg **fr**D,**fr**B (Rc = 0)
fneg. **fr**D,**fr**B (Rc = 1)



The contents of register **fr**B with bit 0 inverted are placed into **fr**D.

Note that the **fneg** instruction treats NaNs just like any other kind of value. That is, the sign bit of a NaN may be altered by **fneg**. This instruction does not alter the FPSCR.

Other registers altered:

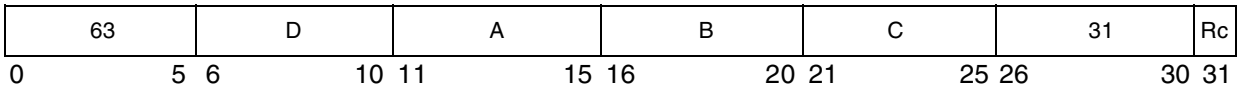
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

fnmadd*x* fnmadd*x*

Floating Negative Multiply-Add (Double-Precision)

fnmadd **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fnmadd. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← – ([frA * frC] + frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add (**fmadd***x*) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

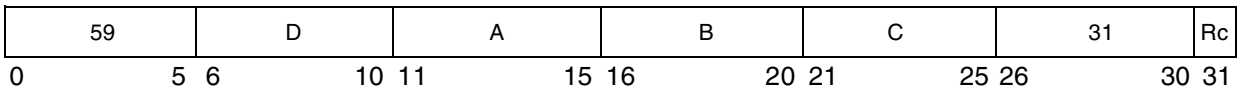
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fnmadds*x* fnmadds*x*

Floating Negative Multiply-Add Single

fnmadds **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fnmadds. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← – ([frA * frC] + frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is added to this intermediate result. If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result as would be obtained by using the Floating Multiply-Add Single (**fmadds***x*) instruction and then negating the result, with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

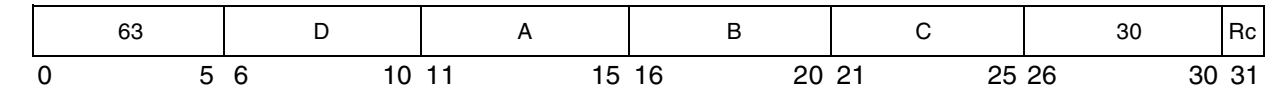
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fnmsub*x* fnmsub*x*

Floating Negative Multiply-Subtract (Double-Precision)

fnmsub **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fnmsub. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← – ([frA * frC] – frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the msb of the resultant significand is not one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract (**fmsub***x*) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

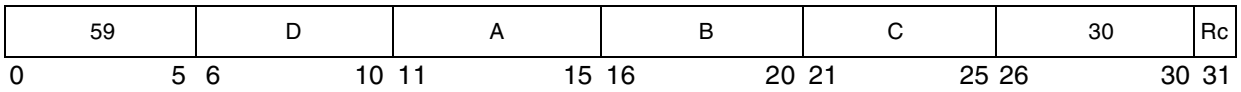
- Condition register (CR1 field)
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fnmsubs*x* fnmsubs*x*

Floating Negative Multiply-Subtract Single

fnmsubs **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fnmsubs. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)



The following operation is performed:

```
frD ← – ([frA * frC] – frB)
```

The floating-point operand in register **fr**A is multiplied by the floating-point operand in register **fr**C. The floating-point operand in register **fr**B is subtracted from this intermediate result.

If the msb of the resultant significand is not one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR, then negated and placed into **fr**D.

This instruction produces the same result obtained by negating the result of a Floating Multiply-Subtract Single (**fmsubs***x*) instruction with the following exceptions:

- QNaNs propagate with no effect on their sign bit.
- QNaNs that are generated as the result of a disabled invalid operation exception have a sign bit of zero.
- SNaNs that are converted to QNaNs as the result of a disabled invalid operation exception retain the sign bit of the SNaN.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition register (CR1 field)
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI, VXIMZ

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fres*x* fres*x*

Floating Reciprocal Estimate Single

fres **fr**D,**fr**B (Rc = 0)
fres. **fr**D,**fr**B (Rc = 1)

A single-precision estimate of the reciprocal of the **fr**B value is placed in **fr**D. The estimate placed into **fr**D is correct to a precision of 1 part in 256 of the reciprocal of **fr**B. That is,

$$\mathrm{ABS}\left(\frac{\text{estimate}-\left(\frac{1}{x}\right)}{\left(\frac{1}{x}\right)}\right) \leq \frac{1}{256}$$

where x is the initial value in **fr**B. Note that the value placed into register **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized below:

Operand	Result	Exception
–∞	–0	None
–0	–∞*	ZX
+0	+∞*	ZX
+∞	+0	None
SNaN	QNaN**	VXSNAN
QNaN	QNaN	None

Notes: * No result if FPSCR[ZE] = 1

** No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1.

Note that the architecture makes no provision for a double-precision version of **fres***x*.

Other registers altered:

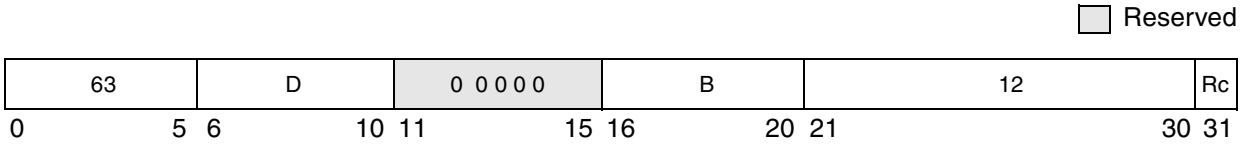
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR (undefined), FI (undefined), FX, OX, UX, ZX, VXSNAN

Architecture Level	Supervisor Level	Optional	Form
UISA		√	A

frsp*x* frsp*x*

Floating Round to Single

frsp **fr**D,**fr**B (Rc = 0)
frsp. **fr**D,**fr**B (Rc = 1)



The floating-point operand in register **fr**B is rounded to single-precision using the rounding mode specified by FPSCR[RN] and placed into **fr**D.

The rounding is described fully in Section C.4.1, “Floating-Point Round to Single-Precision Model.”

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

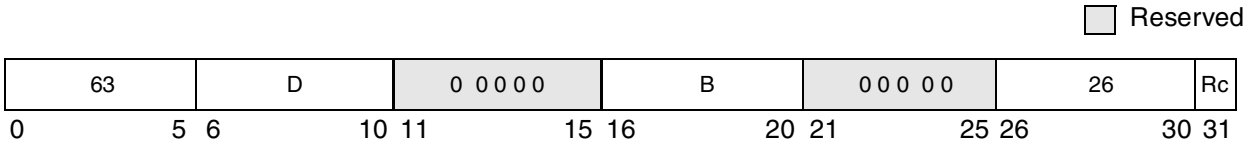
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN

Architecture Level	Supervisor Level	Optional	Form
UISA			X

frsqrte*x* frsqrte*x*

Floating Reciprocal Square Root Estimate

frsqrte **fr**D,**fr**B (Rc = 0)
frsqrte. **fr**D,**fr**B (Rc = 1)



A double-precision estimate of the reciprocal of the square root of the **fr**B value is placed into **fr**D. The estimate is correct to a precision of 1 part in 32. That is,

$$\mathrm{ABS}\left(\frac{\mathrm{estimate}-\left(\frac{1}{\sqrt{x}}\right)}{\left(\frac{1}{\sqrt{x}}\right)}\right) \le \frac{1}{32}$$

where x is the initial value in **fr**B. Note that the value placed into **fr**D may vary between implementations, and between different executions on the same implementation.

Operation with various special values of the operand is summarized below:

Operand	Result	Exception
–∞	QNaN**	VXSQRT
<0	QNaN**	VXSQRT
–0	–∞*	ZX
+0	+∞*	ZX
+∞	+0	None
SNaN	QNaN**	VXSNAN
QNaN	QNaN	None

Notes: * No result if FPSCR[ZE] = 1
** No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1 and zero divide exceptions when FPSCR[ZE] = 1. No single-precision version of the **frsqrte** instruction is provided; however, both **fr**B and **fr**D are representable in single-precision format.

Other registers altered:

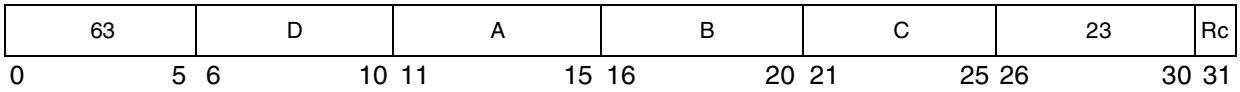
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX(if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR (undefined), FI (undefined), FX, ZX, VXSNAN, VXSQRT

Architecture Level	Supervisor Level	Optional	Form
UISA		√	A

fsel*x* fsel*x*

Floating Select

fsel **fr**D,**fr**A,**fr**C,**fr**B (Rc = 0)
fsel. **fr**D,**fr**A,**fr**C,**fr**B (Rc = 1)




```
if (frA) ≥ 0.0 then frD ← (frC)
else frD ← (frB)
```

The floating-point operand in register **fr**A is compared to the value zero. If the operand is greater than or equal to zero, register **fr**D is set to the contents of register **fr**C. If the operand is less than zero or is a NaN, register **fr**D is set to the contents of register **fr**B. The comparison ignores the sign of zero (that is, regards +0 as equal to –0).

Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

For examples of uses of this instruction, see Section C.3, “Floating-Point Conversions,” and Section C.5, “Floating-Point Selection.”

This instruction is optional in the architecture.

Other registers altered:

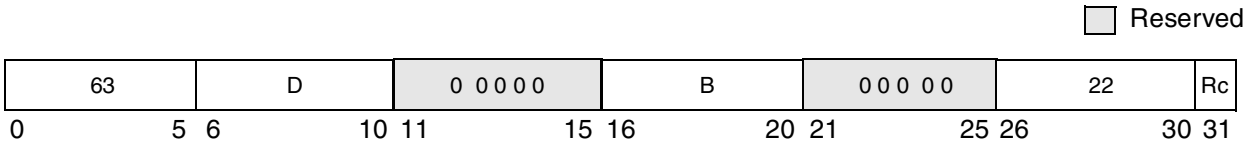
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA		√	A

fsqrt*x* fsqrt*x*

Floating Square Root (Double-Precision)

fsqrt **fr**D,**fr**B (Rc = 0)
fsqrt. **fr**D,**fr**B (Rc = 1)



The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the msb of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D.

Operation with various special values of the operand is summarized below:

Operand	Result	Exception
–∞	QNaN*	VXSQRT
<0	QNaN*	VXSQRT
–0	–0	None
+∞	+∞	None
SNaN	QNaN*	VXSNAN
QNaN	QNaN	None

Notes: * No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

This instruction is optional in the architecture.

Other registers altered:

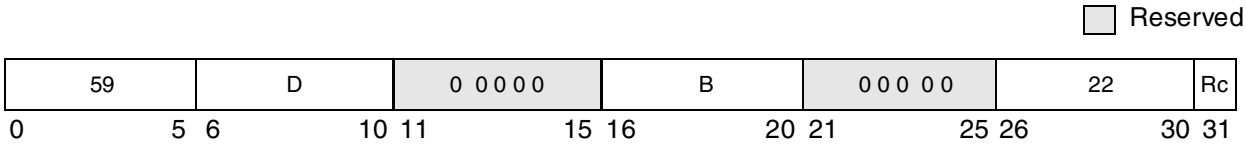
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, XX, VXSNAN, VXSQRT

Architecture Level	Supervisor Level	Optional	Form
UISA		√	A

fsqrts*x* fsqrts*x*

Floating Square Root Single

fsqrts **fr**D,**fr**B (Rc = 0)
fsqrts. **fr**D,**fr**B (Rc = 1)



The square root of the floating-point operand in register **fr**B is placed into register **fr**D.

If the msb of the resultant significand is not a one the result is normalized. The result is rounded to the target precision under control of the floating-point rounding control field RN of the FPSCR and placed into register **fr**D.

Operation with various special values of the operand is summarized below.

Operand	Result	Exception
–∞	QNaN*	VXSQRT
<0	QNaN*	VXSQRT
–0	–0	None
+∞	+∞	None
SNaN	QNaN*	VXSNAN
QNaN	QNaN	None

Notes: * No result if FPSCR[VE] = 1

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

This instruction is optional in the architecture.

Other registers altered:

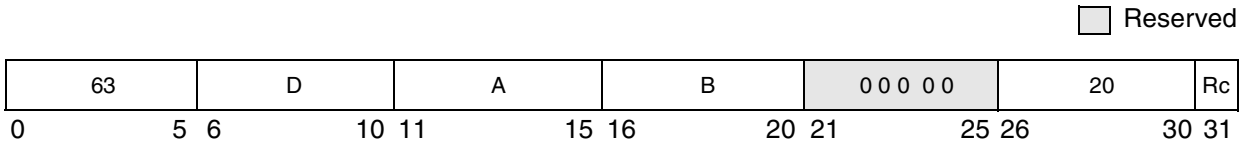
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, XX, VXSNAN, VXSQRT

Architecture Level	Supervisor Level	Optional	Form
UISA		√	A

fsub*x* fsub*x*

Floating Subtract (Double-Precision)

fsub **fr**D,**fr**A,**fr**B (Rc = 0)
fsub. **fr**D,**fr**A,**fr**B (Rc = 1)



The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

The execution of the **fsub** instruction is identical to that of **fadd**, except that the contents of **fr**B participate in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

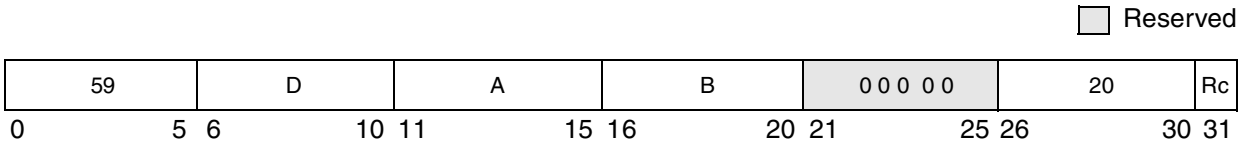
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

Architecture Level	Supervisor Level	Optional	Form
UISA			A

fsubs*x* fsubs*x*

Floating Subtract Single

fsubs **fr**D,**fr**A,**fr**B (Rc = 0)
fsubs. **fr**D,**fr**A,**fr**B (Rc = 1)



The floating-point operand in register **fr**B is subtracted from the floating-point operand in register **fr**A. If the msb of the resultant significand is not a one, the result is normalized. The result is rounded to single-precision under control of the floating-point rounding control field RN of the FPSCR and placed into **fr**D.

The execution of the **fsubs** instruction is identical to that of **fadds**, except that the contents of **fr**B participate in the operation with its sign bit (bit 0) inverted.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Other registers altered:

- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPRF, FR, FI, FX, OX, UX, XX, VXSNAN, VXISI

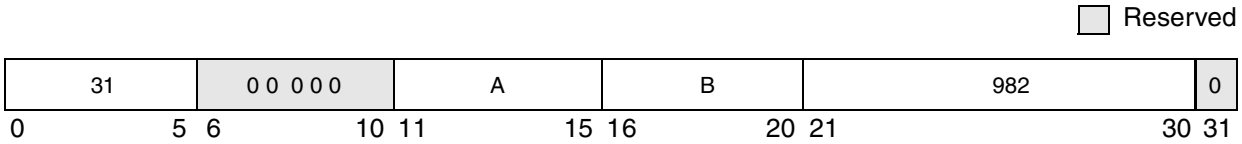
Architecture Level	Supervisor Level	Optional	Form
UISA			A

icbi

icbi

Instruction Cache Block Invalidate

icbi **r**A,**r**B



EA is the sum (**r**A|0) + (**r**B).

If the block containing the byte addressed by EA is in coherency-required mode, and a block containing the byte addressed by EA is in the instruction cache of any processor, the block is made invalid in all such instruction caches, so that subsequent references cause the block to be refetched.

If the block containing the byte addressed by EA is in coherency-not-required mode, and a block containing the byte addressed by EA is in the instruction cache of this processor, the block is made invalid in that instruction cache, so that subsequent references cause the block to be refetched.

The function of this instruction is independent of the write-through, write-back, and caching-inhibited/allowed modes of the block containing the byte addressed by EA.

This instruction is treated as a load from the addressed byte with respect to address translation and memory protection. It may also be treated as a load for reference and change bit recording except that reference and change bit recording may not occur. Implementations with a combined data and instruction cache treat the **icbi** instruction as a no-op, except that they may invalidate the target block in the instruction caches of other processors if the block is in coherency-required mode.

The **icbi** instruction invalidates the block at EA (**r**A|0 + **r**B). If the processor is a multiprocessor implementation and the block is marked coherency-required, the processor will send an address-only broadcast to other processors causing those processors to invalidate the block from their instruction caches.

For faster processing, many implementations will not compare the entire EA (**r**A|0 + **r**B) with the tag in the instruction cache. Instead, they will use the bits in the EA to locate the set that the block is in, and invalidate all blocks in that set.

Other registers altered:

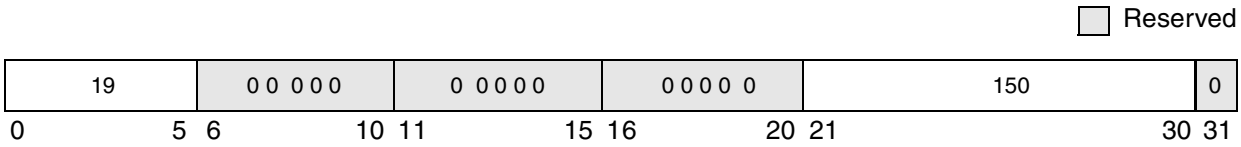
- None

Architecture Level	Supervisor Level	Optional	Form
VEA			X

isync — isync

Instruction Synchronize

isync



The **isync** instruction provides an ordering function for the effects of all instructions executed by a processor. Executing an **isync** instruction ensures that all instructions preceding the **isync** instruction have completed before the **isync** instruction completes, except that memory accesses caused by those instructions need not have been performed with respect to other processors and mechanisms. It also ensures that no subsequent instructions are initiated by the processor until after the **isync** instruction completes. Finally, it causes the processor to discard any prefetched instructions, with the effect that subsequent instructions will be fetched and executed in the context established by the instructions preceding the isync instruction. The **isync** instruction has no effect on the other processors or on their caches.

This instruction is context synchronizing.

Context synchronization is necessary after certain code sequences that perform complex operations within the processor. These code sequences are usually operating system tasks that involve memory management. For example, if an instruction A changes the memory translation rules in the memory management unit (MMU), the **isync** instruction should be executed so that the instructions following instruction A will be discarded from the pipeline and refetched according to the new translation rules.

Note that all interrupts and the **rfi** instruction are also context synchronizing.

Other registers altered:

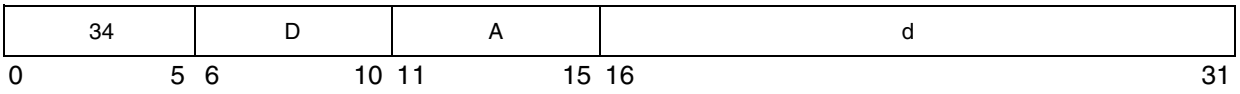
- None

Architecture Level	Supervisor Level	Optional	Form
VEA			XL

lbz lbz

Load Byte and Zero

lbz **r**D,d(**r**A)




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + EXTS(d)
rD ← (24)0 || MEM(EA, 1)
```

EA is the sum (**r**A|0) + d. The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

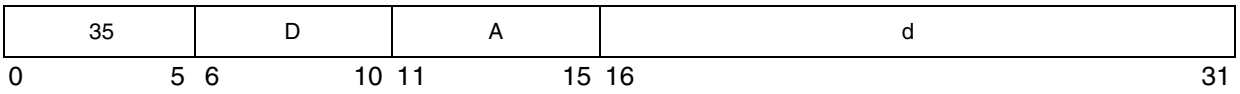
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lbzu lbzu

Load Byte and Zero with Update

lbzu **r**D,d(**r**A)




```
EA ← (rA) + EXTS(d)
rD ← (24)0 || MEM(EA, 1)
rA ← EA
```

EA is the sum (**r**A) + d. The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

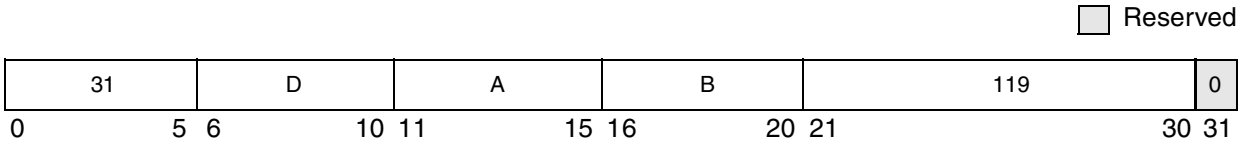
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lbzux lbzux

Load Byte and Zero with Update Indexed

lbzux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD ← (24)0 || MEM(EA, 1)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

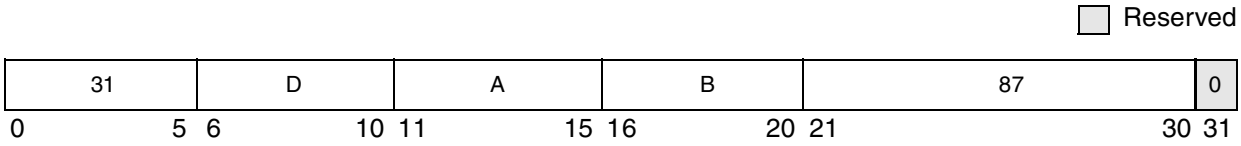
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lbzx lbzx

Load Byte and Zero Indexed

lbzx **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← (24)0 || MEM(EA, 1)
```

EA is the sum (**r**A|0) + (**r**B). The byte in memory addressed by EA is loaded into the low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

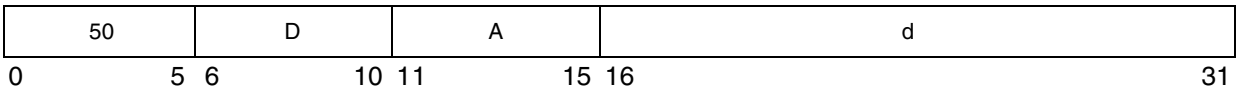
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lfd lfd

Load Floating-Point Double

lfd **fr**D,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
frD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + d.

The double word in memory addressed by EA is placed into **fr**D.

Other registers altered:

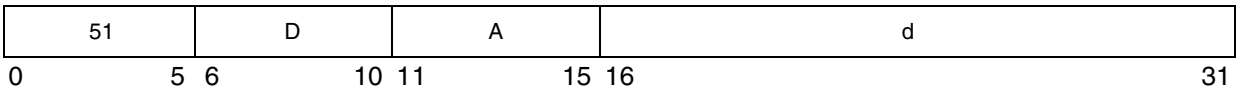
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lfdu lfdu

Load Floating-Point Double with Update

lfdu **fr**D,d(**r**A)




```
EA ← (rA) + EXTS(d)
frD ← MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + d.

The double word in memory addressed by EA is placed into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

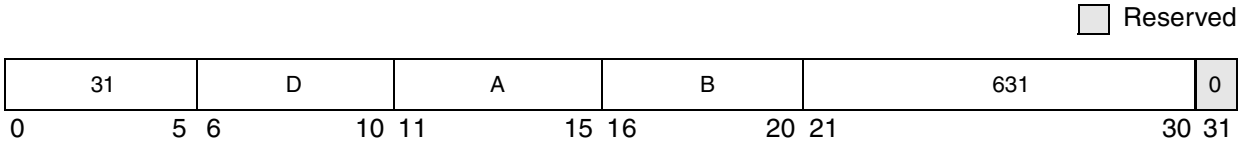
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lfdux lfdux

Load Floating-Point Double with Update Indexed

lfdux **fr**D,**r**A,**r**B




```
EA ← (rA) + (rB)
frD ← MEM(EA, 8)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The double word in memory addressed by EA is placed into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

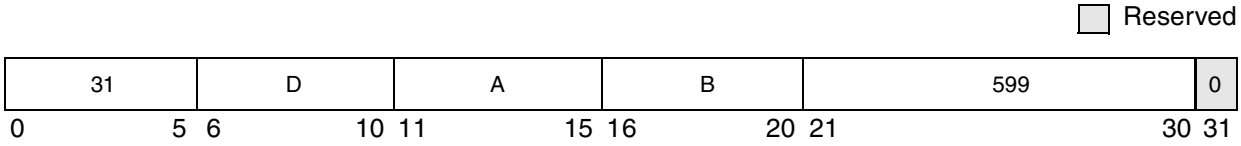
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lfdx lfdx

Load Floating-Point Double Indexed

lfdx **fr**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
frD ← MEM(EA, 8)
```

EA is the sum (**r**A|0) + (**r**B).

The double word in memory addressed by EA is placed into **fr**D.

Other registers altered:

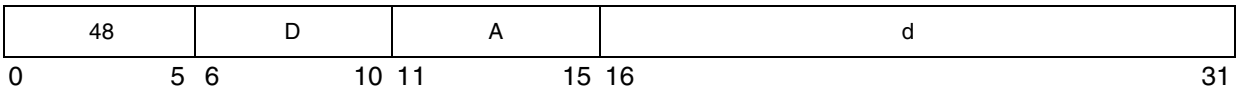
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lfs lfs

Load Floating-Point Single

lfs **fr**D,d(**r**A)




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + EXTS(d)
frD ← DOUBLE(MEM(EA, 4))
```

EA is the sum (**r**A|0) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see Section C.6, “Floating-Point Load Instructions”) and placed into **fr**D.

Other registers altered:

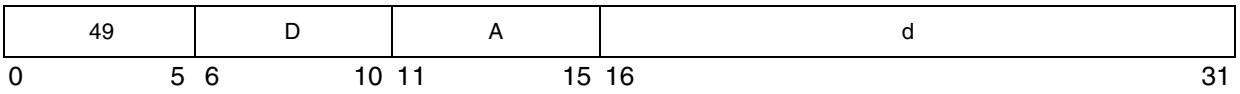
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lfsu lfsu

Load Floating-Point Single with Update

lfsu **fr**D,d(**r**A)




```
EA ← (rA) + EXTS(d)
frD ← DOUBLE(MEM(EA, 4))
rA ← EA
```

EA is the sum (**r**A) + d.

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see Section C.6, “Floating-Point Load Instructions”) and placed into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

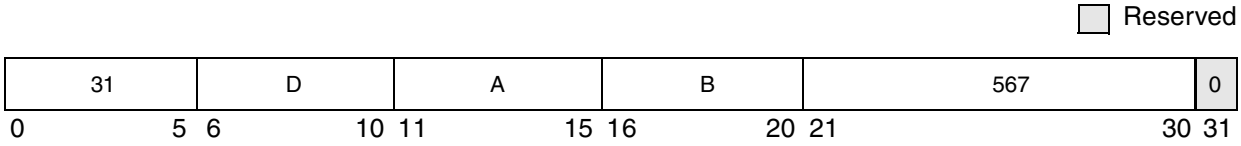
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lfsux lfsux

Load Floating-Point Single with Update Indexed

lfsux **fr**D,**r**A,**r**B




```
EA ← (rA) + (rB)
frD ← DOUBLE(MEM(EA, 4))
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see Section C.6, “Floating-Point Load Instructions”) and placed into **fr**D.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

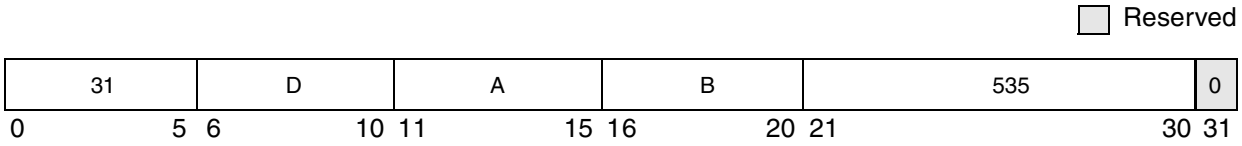
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lfsx lfsx

Load Floating-Point Single Indexed

lfsx **fr**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
frD ← DOUBLE(MEM(EA, 4))
```

EA is the sum (**r**A|0) + (**r**B).

The word in memory addressed by EA is interpreted as a floating-point single-precision operand. This word is converted to floating-point double-precision (see Section C.6, “Floating-Point Load Instructions”) and placed into **fr**D.

Other registers altered:

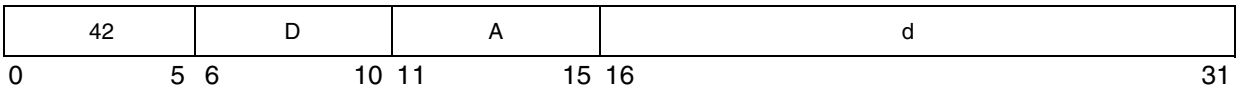
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lha lha

Load Half Word Algebraic

lha **r**D,d(**r**A)




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + EXTS(d)
rD ← EXTS(MEM(EA, 2))
```

EA is the sum (**r**A|0) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the msb of the loaded half word.

Other registers altered:

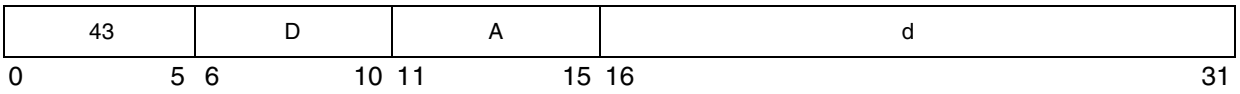
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lhau lhau

Load Half Word Algebraic with Update

lhau **r**D,d(**r**A)




```
EA ← (rA) + EXTS(d)
rD ← EXTS(MEM(EA, 2))
rA ← EA
```

EA is the sum (**r**A) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the msb of the loaded half word.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

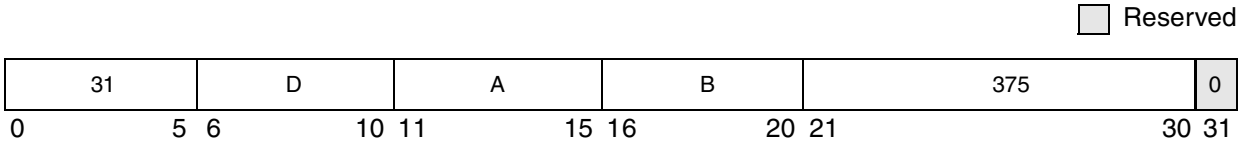
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lhaux lhaux

Load Half Word Algebraic with Update Indexed

lhaux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD ← EXTS(MEM(EA, 2))
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the msb of the loaded half word.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

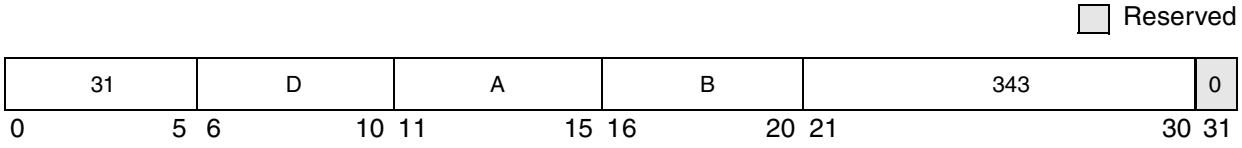
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lhax lhax

Load Half Word Algebraic Indexed

lhax **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← EXTS(MEM(EA, 2))
```

EA is the sum (**r**A|0) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are filled with a copy of the msb of the loaded half word.

Other registers altered:

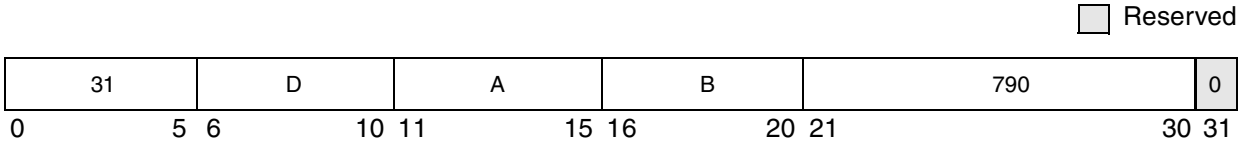
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lhbrx lhbrx

Load Half Word Byte-Reverse Indexed

lhbrx **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
rD ← (16)0 || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + (**r**B). Bits 0–7 of the half word in memory addressed by EA are loaded into the low-order eight bits of **r**D. Bits 8–15 of the half word in memory addressed by EA are loaded into the subsequent low-order eight bits of **r**D. The remaining bits in **r**D are cleared.

The architecture cautions programmers that some implementations of the architecture may run the **lhbrx** instructions with greater latency than other types of load instructions.

Other registers altered:

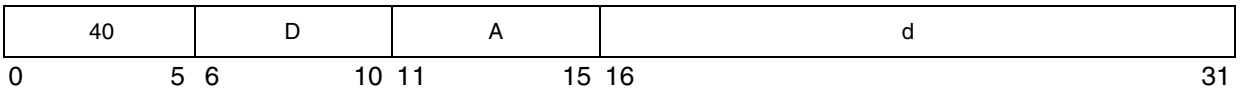
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lhz lhz

Load Half Word and Zero

lhz **r**D,d(**r**A)




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + EXTS(d)
rD ← (16)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

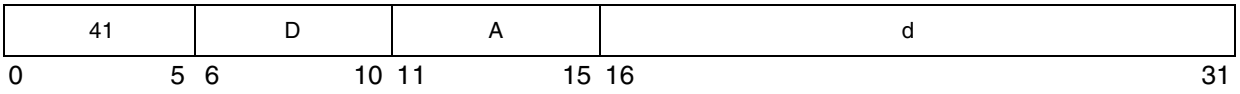
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lhzu lhzu

Load Half Word and Zero with Update

lhzu **r**D,d(**r**A)




```
EA ← rA + EXTS(d)
rD ← (16)0 || MEM(EA, 2)
rA ← EA
```

EA is the sum (**r**A) + d. The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

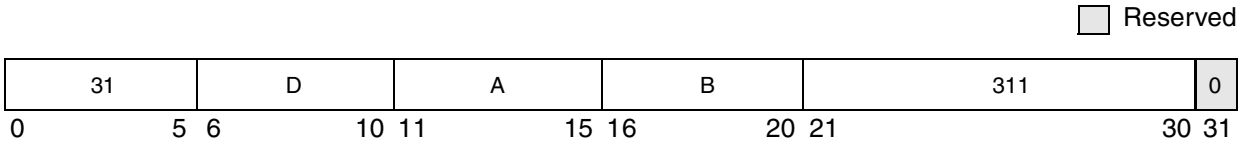
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lhzux lhzux

Load Half Word and Zero with Update Indexed

lhzux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD ← (16)0 || MEM(EA, 2)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

EA is placed into **r**A.

If **r**A = 0 or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

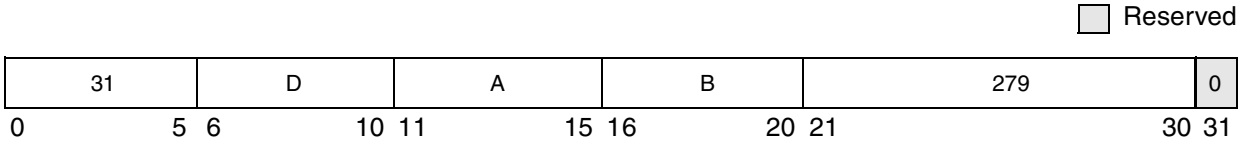
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lhzx lhzx

Load Half Word and Zero Indexed

lhzx **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
rD ← (16)0 || MEM(EA, 2)
```

EA is the sum (**r**A|0) + (**r**B). The half word in memory addressed by EA is loaded into the low-order 16 bits of **r**D. The remaining bits in **r**D are cleared.

Other registers altered:

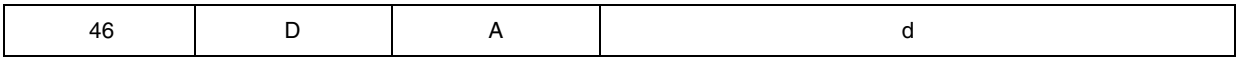
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lmw lmw

Load Multiple Word

lmw **r**D,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
r ← rD
do while r ≤ 31
   GPR(r) ← MEM(EA, 4)
   r ← r + 1
   EA ← EA + 4
```

EA is the sum (**r**A|0) + d.

n = (32 – **r**D).

n consecutive words starting at EA are loaded into GPRs **r**D through **r31**.

EA must be a multiple of four. If it is not, either the system alignment interrupt handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI interrupts, see Section 6.5.3, “Data Storage Interrupt (0x00300).”

If **r**A is in the range of registers specified to be loaded, including the case in which **r**A = 0, the instruction form is invalid.

Note that, in some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

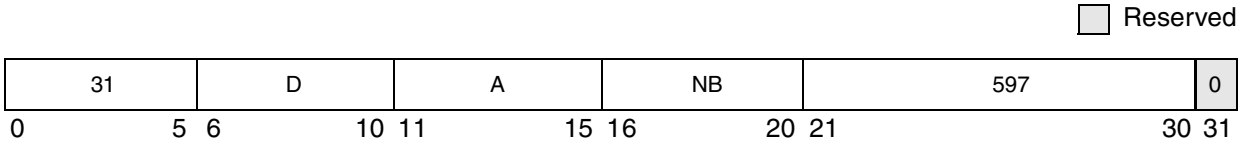
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lswi lswi

Load String Word Immediate

lswi **r**D,**r**A,NB




```
if rA = 0 then EA ← 0
else EA ← (rA)
if NB = 0 then n ← 32
else n ← NB
r ← rD - 1
i ← 0
do while n > 0
   if i = 32 then
     r ← r + 1 (mod 32)
     GPR(r) ← 0
   GPR(r)[i–i + 7] ← MEM(EA, 1)
   i ← i + 8
   if i = 32 then i ← 0
   EA ← EA + 1
   n ← n - 1
```

EA is (**r**A | 0).
Let *n* = NB if NB ≠ 0, *n* = 32 if NB = 0; *n* is the number of bytes to load.
Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to be loaded with data.
n consecutive bytes starting at EA are loaded into GPRs **r**D through **r**D + *nr* – 1.

Bytes are loaded left to right in each register. The sequence of registers wraps around to **r0** if required. If the 4 bytes of register **r**D + *nr* – 1 are only partially filled, the unfilled low-order byte(s) of that register are cleared.

If **r**A is in the range of registers specified to be loaded, including the case in which **r**A = 0, the instruction form is invalid. Under certain conditions (for example, segment boundary crossing) the data alignment interrupt handler may be invoked. See Section 6.5.3, “Data Storage Interrupt (0x00300).”

In some implementations this instruction is likely to have greater latency than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

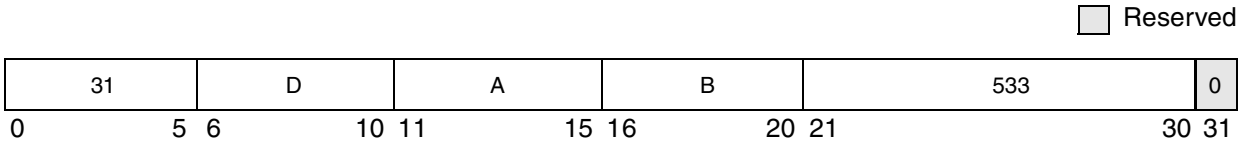
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lswx lswx

Load String Word Indexed

lswx **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
n ← XER[25:31]
r ← rD – 1
i ← 32
rD ← undefined
 do while n > 0
  if i = 32 then
    r ← r + 1 (mod 32)
    GPR(r) ← 0
  GPR(r)[i–i + 7] ← MEM(EA, 1)
  i ← i + 8
  if i = 32 then i ← 0
  EA ← EA + 1
  n ← n – 1
```

EA is the sum (**r**A|0) + (**r**B). Let *n* = XER[25–31]; *n* is the number of bytes to load. Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to receive data. If *n* > 0, *n* consecutive bytes starting at EA are loaded into GPRs **r**D through **r**D + *nr* – 1.

Bytes are loaded left to right in each register. The register sequence wraps around through **r0** if required. If the 4 bytes of **r**D + *nr* – 1 are only partially filled, the unfilled low-order bytes of that register are cleared. If *n* = 0, the contents of **r**D are undefined. If **r**A or **r**B is in the range of registers specified to be loaded, including **r**A = 0, either the system illegal instruction error handler is invoked or results are boundedly undefined. If **r**D = **r**A or **r**D = **r**B, the instruction form is invalid. If **r**D and **r**A both specify GPR0, the form is invalid.

Under some conditions (for example, segment boundary crossing) the data alignment interrupt handler may be invoked. See Section 6.5.3, “Data Storage Interrupt (0x00300).”

In some implementations, **lswx** is likely to take much longer to execute than a sequence of individual load or store instructions that produce the same results.

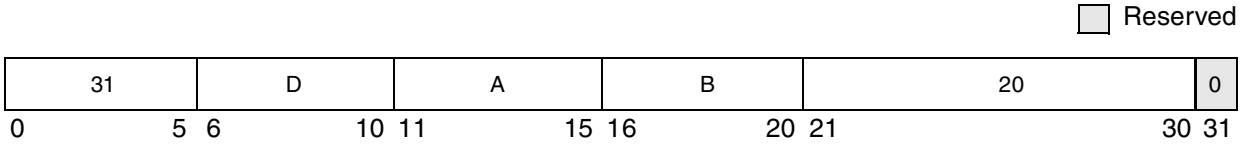
Other registers altered: None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lwarx lwarx

Load Word and Reserve Indexed

lwarx **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
RESERVE ← 1
RESERVE_ADDR ← physical_addr(EA)
rD ← MEM(EA,4)
```

EA is the sum (**r**A|0) + (**r**B).

The word in memory addressed by EA is loaded into **r**D.

This instruction creates a reservation for use by a store word conditional indexed (**stwcx.**)instruction. The physical address computed from EA is associated with the reservation, and replaces any address previously associated with the reservation.

EA must be a multiple of four. If it is not, either the system alignment interrupt handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI interrupts, see Section 6.5.3, “Data Storage Interrupt (0x00300).”

When the RESERVE bit is set, the processor enables hardware snooping for the block of memory addressed by the RESERVE address. If the processor detects that another processor writes to the block of memory it has reserved, it clears the RESERVE bit. The **stwcx.** instruction will only do a store if the RESERVE bit is set. The **stwcx.** instruction sets the CR0[EQ] bit if the store was successful and clears it if it failed. The **lwarx** and **stwcx.** combination can be used for atomic read-modify-write sequences. Note that the atomic sequence is not guaranteed, but its failure can be detected if CR0[EQ] = 0 after the **stwcx.** instruction.

Other registers altered:

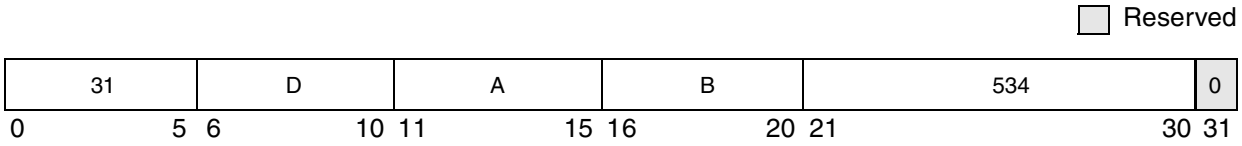
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lwbrx lwbrx

Load Word Byte-Reverse Indexed

lwbrx **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
rD ← MEM(EA + 3, 1) || MEM(EA + 2, 1) || MEM(EA + 1, 1) || MEM(EA, 1)
```

EA is the sum (**r**A|0) + **r**B. Bits 0–7 of the word in memory addressed by EA are loaded into the low-order 8 bits of **r**D. Bits 8–15 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D. Bits 16–23 of the word in memory addressed by EA are loaded into the subsequent low-order eight bits of **r**D. Bits 24–31 of the word in memory addressed by EA are loaded into the subsequent low-order 8 bits of **r**D.

The architecture cautions programmers that some implementations of the architecture may run the **lwbrx** instructions with greater latency than other types of load instructions.

Other registers altered:

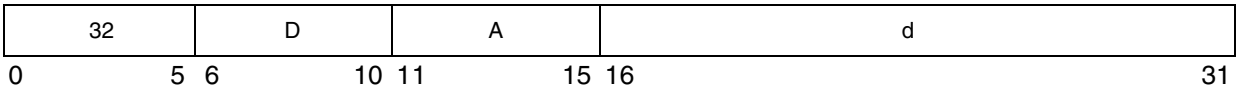
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lwz lwz

Load Word and Zero

lwz **r**D,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
rD ← MEM(EA, 4)
```

EA is the sum (**r**A|0) + d. The word in memory addressed by EA is loaded into **r**D.

Other registers altered:

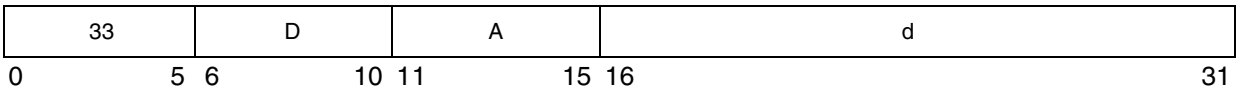
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lwzu lwzu

Load Word and Zero with Update

lwzu **r**D,d(**r**A)




```
EA ← rA + EXTS(d)
rD ← MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + d. The word in memory addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = 0, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

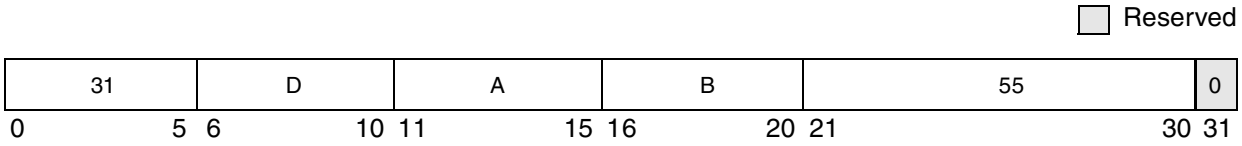
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

lwzux lwzux

Load Word and Zero with Update Indexed

lwzux **r**D,**r**A,**r**B




```
EA ← (rA) + (rB)
rD ← MEM(EA, 4)
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The word in memory addressed by EA is loaded into **r**D.

EA is placed into **r**A.

If **r**A = 0, or **r**A = **r**D, the instruction form is invalid.

Other registers altered:

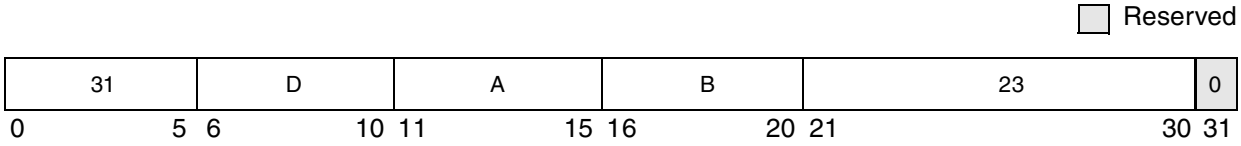
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

lwzx lwzx

Load Word and Zero Indexed

lwzx **r**D,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + rB
rD ← MEM(EA, 4)
```

EA is the sum (**r**A|0) + (**r**B). The word in memory addressed by EA is loaded into **r**D.

Other registers altered:

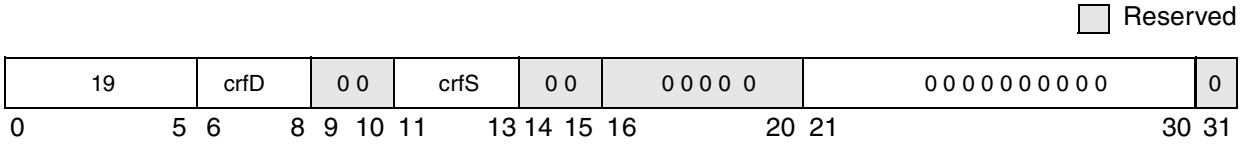
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mcrf mcrf

Move Condition Register Field

mcrf **crf**D,**crf**S




```
CR[(4 * crfD):(4 * crfD + 3)] ← CR[(4 * crfS):(4 * crfS + 3)]
```

The contents of condition register field **crf**S are copied into condition register field **crf**D. All other condition register fields remain unchanged.

Other registers altered:

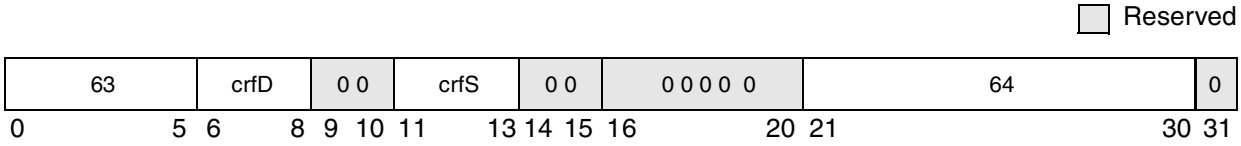
- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO

Architecture Level	Supervisor Level	Optional	Form
UISA			XL

mcrfs mcrfs

Move to Condition Register from FPSCR

mcrfs **crf**D**,crf**S



The contents of FPSCR field **crf**S are copied to CR field **crf**D. All exception bits copied (except FEX and VX) are cleared in the FPSCR.

Other registers altered:

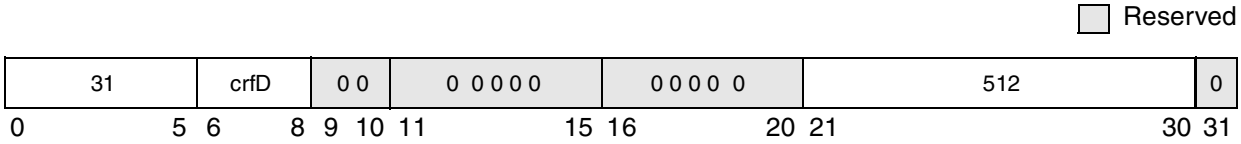
- Condition register (CR field specified by operand **crf**D):
  Affected: FX, FEX, VX, OX
- Floating-point status and control register:
  Affected: FX, OX (if **crf**S = 0)
  Affected: UX, ZX, XX, VXSNAN (if **crf**S = 1)
  Affected: VXISI, VXIDI, VXZDZ, VXIMZ (if **crf**S = 2)
  Affected: VXVC (if **crf**S = 3)
  Affected: VXSOFT, VXSQRT, VXCVI (if **crf**S = 5)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mcrxr mcrxr

Move to Condition Register from XER

mcrxr **crf**D




```
CR[(4 * crfD):(4 * crfD + 3)] ← XER[0:3]
XER[0:3] ← 0b0000
```

The contents of XER[0–3] are copied into the condition register field designated by **crf**D. All other fields of the condition register remain unchanged. XER[0–3] is cleared.

Other registers altered:

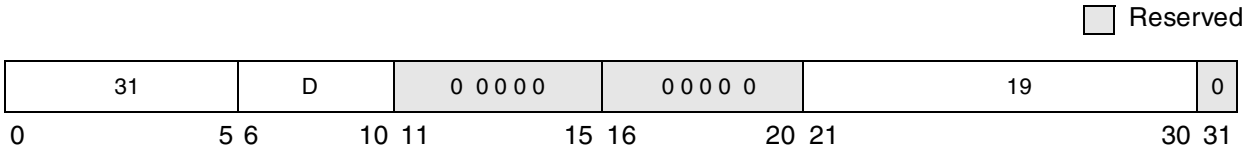
- Condition register (CR field specified by operand **crf**D):
  Affected: LT, GT, EQ, SO
- XER[0–3]

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mfcr mfcr

Move from Condition Register

mfcr **r**D



rD ← CR

The contents of the condition register (CR) are placed into **r**D.

Other registers altered:

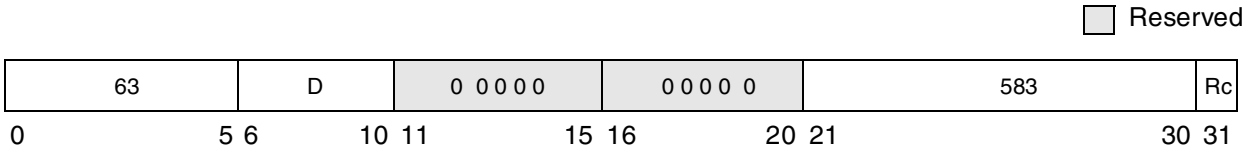
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mffs*x* mffs*x*

Move from FPSCR

mffs **fr**D (Rc = 0)
mffs. **fr**D (Rc = 1)




```
frD[32:63] ← FPSCR
```

The contents of the FPSCR are placed into the low-order 32-bits of register **fr**D. The high-order 32-bits of register **fr**D are undefined.

Other registers altered:

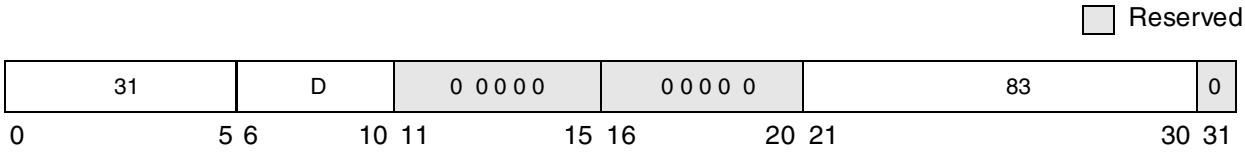
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mfmsr mfmsr

Move from Machine State Register

mfmsr **r**D



rD ← MSR

The contents of the MSR are placed into **r**D.

This is a supervisor-level instruction.

Other registers altered:

- None

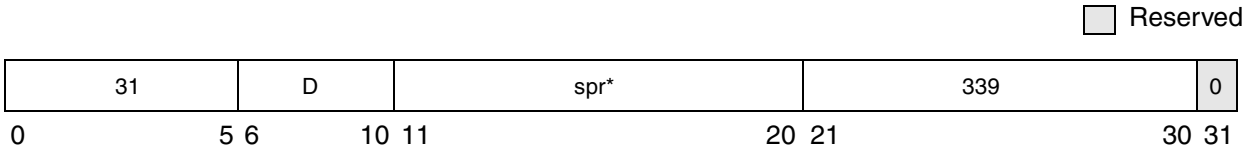
Architecture Level	Supervisor Level	Optional	Form
OEA	√		X

mfspr

mfspr

Move from Special-Purpose Register

mfspr **r**D**,**SPR



***Note:** This is a split field.

```
n ← spr[5:9] || spr[0:4]
  rD ← SPR(n)
```

In the UISA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-12. The contents of the designated special-purpose register are placed into **r**D.

Table 8-12. UISA SPR Encodings for mfspr

SPR**			Register Name
Decimal	spr[5–9]	spr[0–4]	
1	00000	00001	XER
8	00000	01000	LR
9	00000	01001	CTR

** Note that the order of the two 5-bit halves of the SPR number is reversed compared with the actual instruction coding.

If the SPR field contains any value other than one of the values shown in Table 8-12 (and the processor is in user mode), one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The results are boundedly undefined.

Other registers altered:

- None

Simplified mnemonics:

mfxer **r**D	equivalent to	**mfspr** **r**D**,1**
mflr **r**D	equivalent to	**mfspr** **r**D**,8**
mfctr **r**D	equivalent to	**mfspr** **r**D**,9**

In the OEA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-13. The contents of the designated SPR are placed into **r**D.

SPR[0] = 1 if and only if reading the register is supervisor-level. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = 1 will result in a privileged instruction type program interrupt.

If MSR[PR] = 1, the only effect of executing an instruction with an SPR number that is not shown in Table 8-13 and has SPR[0] = 1 is to cause a supervisor-level instruction type program interrupt or an illegal instruction type program interrupt. For all other cases, MSR[PR] = 0 or SPR[0] = 0. If the SPR field contains any value that is not shown in Table 8-13, either an illegal instruction type program interrupt occurs or the results are boundedly undefined.

Other registers altered:

- None

Table 8-13. OEA SPR Encodings for mfspr

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1	00000	00001	XER	User
8	00000	01000	LR	User
9	00000	01001	CTR	User
18	00000	10010	DSISR	Supervisor
19	00000	10011	DAR	Supervisor
22	00000	10110	DEC	Supervisor
25	00000	11001	SDR1	Supervisor
26	00000	11010	SRR0	Supervisor
27	00000	11011	SRR1	Supervisor
272	01000	10000	SPRG0	Supervisor
273	01000	10001	SPRG1	Supervisor
274	01000	10010	SPRG2	Supervisor
275	01000	10011	SPRG3	Supervisor

Table 8-13. OEA SPR Encodings for mfspr (continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
282	01000	11010	EAR	Supervisor
287	01000	11111	PVR	Supervisor
528	10000	10000	IBAT0U	Supervisor
529	10000	10001	IBAT0L	Supervisor
530	10000	10010	IBAT1U	Supervisor
531	10000	10011	IBAT1L	Supervisor
532	10000	10100	IBAT2U	Supervisor
533	10000	10101	IBAT2L	Supervisor
534	10000	10110	IBAT3U	Supervisor
535	10000	10111	IBAT3L	Supervisor
536	10000	11000	DBAT0U	Supervisor
537	10000	11001	DBAT0L	Supervisor
538	10000	11010	DBAT1U	Supervisor
539	10000	11011	DBAT1L	Supervisor
540	10000	11100	DBAT2U	Supervisor
541	10000	11101	DBAT2L	Supervisor
542	10000	11110	DBAT3U	Supervisor
543	10000	11111	DBAT3L	Supervisor
1013	11111	10101	DABR	Supervisor

[1]Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits 16–20 of the instruction and the low-order five bits in bits 11–15.

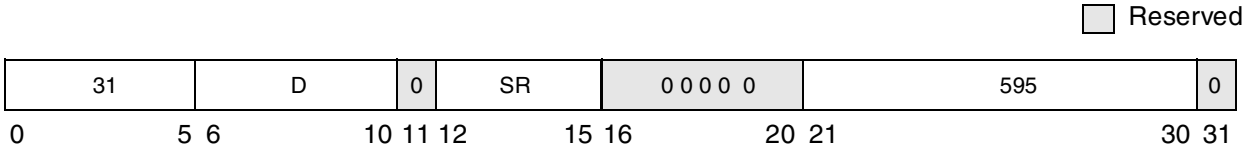
Architecture Level	Supervisor Level	Optional	Form
UISA/OEA	√*		XFX

* Note that **mfspr** is supervisor-level only if SPR[0] = 1.

mfsr mfsr

Move from Segment Register

mfsr **r**D,SR




```
rD ← SEGREG(SR)
```

The contents of segment register SR are placed into **r**D.

This is a supervisor-level instruction.

Other registers altered:

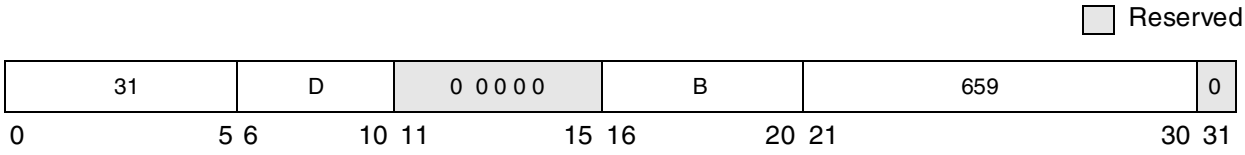
- None

Architecture Level	Supervisor Level	Optional	Form
OEA	√		X

mfsrin mfsrin

Move from Segment Register Indirect

mfsrin **r**D,**r**B




```
rD ← SEGREG(rB[0:3])
```

The contents of the segment register selected by bits 0–3 of **r**B are copied into **r**D.

This is a supervisor-level instruction.

Note that the **r**A field is not defined for the **mfsrin** instruction in the architecture.

Other registers altered:

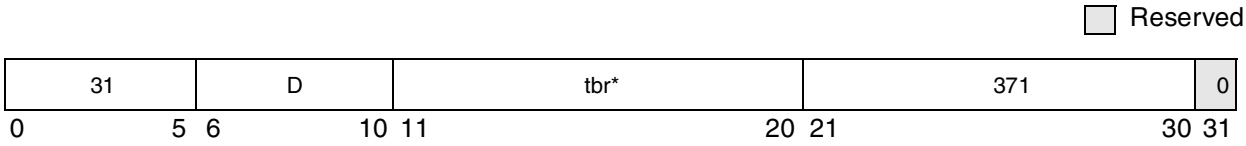
- None

Architecture Level	Supervisor Level	Optional	Form
OEA	√		X

mftb mftb

Move from Time Base

mftb **r**D,TBR



***Note:** This is a split field.

```
n ← tbr[5:9] || tbr[0:4]
if n = 268 then
         rD ← TBL
else if n = 269 then
         rD ← TBU
```

The contents of TBL or TBU are copied into rD, as designated by the value in TBR, encoded as shown in Table 8-14.

Table 8-14. TBR Encodings for mftb

TBR*			Register Name	Access
Decimal	tbr[5–9]	tbr[0–4]		
268	01000	01100	TBL	User
269	01000	01101	TBU	User

*Note that the order of the two 5-bit halves of the TBR number is reversed.

If the TBR contains a value not shown in Table 8-14, one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor-level instruction error handler is invoked.
- The results are boundedly undefined.

Some implementations may implement **mftb** and **mfspr** identically. See Section 2.2, “VEA Register Set—Time Base.”

Other registers altered:

- None

Simplified mnemonics:

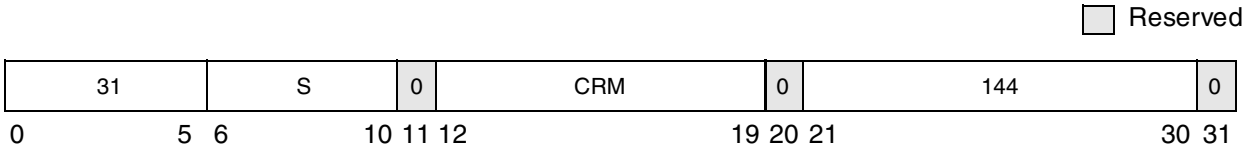
mftb **r**D	equivalent to	**mftb** **r**D**,268**
mftbu **r**D	equivalent to	**mftb** **r**D**,269**

Architecture Level	Supervisor Level	Optional	Form
VEA			XFX

mtcrf mtcrf

Move to Condition Register Fields

mtcrf CRM**,r**S




```
mask ← (4)(CRM[0]) || (4)(CRM[1]) ||... (4)(CRM[7])
CR ← (rS & mask) | (CR & ¬ mask)
```

The contents of **r**S are placed into the condition register under control of the field mask specified by CRM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If CRM(i) = 1, CR field i (CR bits 4 ∗ i through 4 ∗ i + 3) is set to the contents of the corresponding field of **r**S.

Note that updating a subset of the eight fields of the condition register may have substantially poorer performance on some implementations than updating all of the fields.

Other registers altered:

- CR fields selected by mask

Simplified mnemonics:

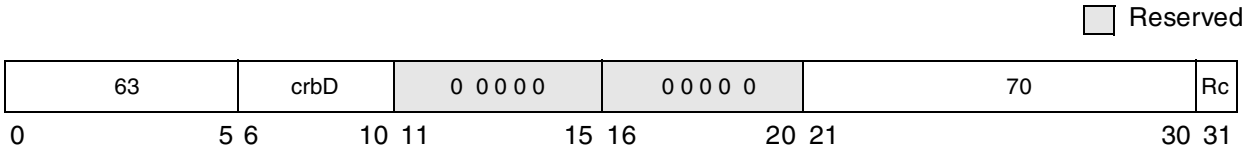
mtcr **r**S equivalent to **mtcrf** 0xFF**,r**S

Architecture Level	Supervisor Level	Optional	Form
UISA			XFX

mtfsb0*x* mtfsb0*x*

Move to FPSCR Bit 0

mtfsb0 **crb**D (Rc = 0)
mtfsb0. **crb**D (Rc = 1)



Bit **crb**D of the FPSCR is cleared.

Other registers altered:

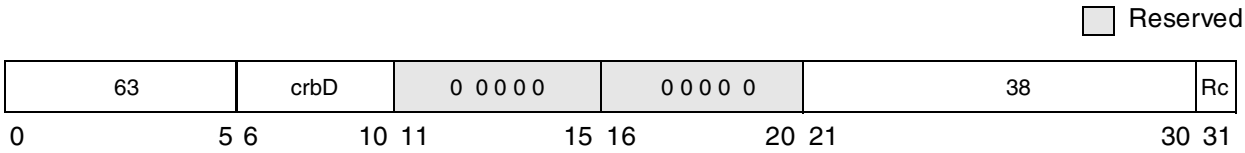
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPSCR bit **crb**D
  Note: Bits 1 and 2 (FEX and VX) cannot be explicitly cleared.

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mtfsb1*x* mtfsb1*x*

Move to FPSCR Bit 1

mtfsb1 **crb**D (Rc = 0)
mtfsb1. **crb**D (Rc = 1)



Bit **crb**D of the FPSCR is set.

Other registers altered:

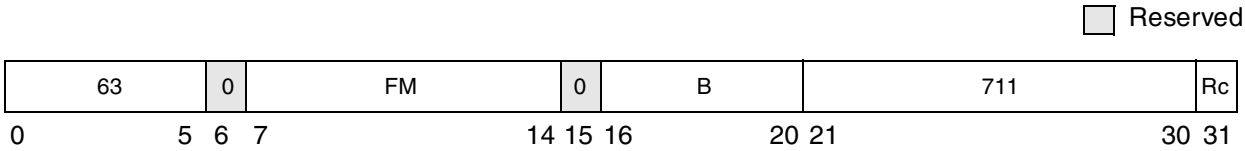
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPSCR bit **crb**D and FX
  Note: Bits 1 and 2 (FEX and VX) cannot be explicitly set.

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mtfsf*x* mtfsf*x*

Move to FPSCR Fields

mtfsf FM,**fr**B (Rc = 0)
mtfsf. FM,**fr**B (Rc = 1)



The low-order 32 bits of **fr**B are placed into the FPSCR under control of the field mask specified by FM. The field mask identifies the 4-bit fields affected. Let i be an integer in the range 0–7. If FM[i] = 1, FPSCR field i (FPSCR bits 4 * i through 4 * i + 3) is set to the contents of the corresponding field of the low-order 32 bits of register **fr**B.

FPSCR[FX] is altered only if FM[0] = 1.

Updating fewer than all eight fields of the FPSCR may have substantially poorer performance on some implementations than updating all the fields.

When FPSCR[0–3] is specified, bits 0 (FX) and 3 (OX) are set to the values of **fr**B[32] and **fr**B[35] (that is, even if this instruction causes OX to change from 0 to 1, FX is set from **fr**B[32] and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule and not from **fr**B[33–34].

Other registers altered:

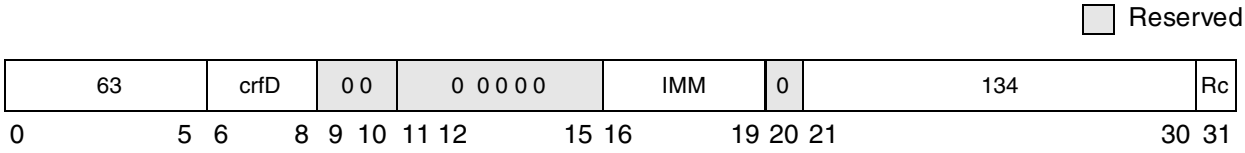
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPSCR fields selected by mask

Architecture Level	Supervisor Level	Optional	Form
UISA			XFL

mtfsfi*x* mtfsfi*x*

Move to FPSCR Field Immediate

mtfsfi **crf**D,IMM (Rc = 0)
mtfsfi. **crf**D,IMM (Rc = 1)




```
FPSCR[crfD] ← IMM
```

The value of the IMM field is placed into FPSCR field **crf**D.

FPSCR[FX] is altered only if **crf**D = 0.

When FPSCR[0–3] is specified, bits 0 (FX) and 3 (OX) are set to the values of IMM[0] and IMM[3] (that is, even if this instruction causes OX to change from 0 to 1, FX is set from IMM[0] and not by the usual rule that FX is set when an exception bit changes from 0 to 1). Bits 1 and 2 (FEX and VX) are set according to the usual rule and not from IMM[1–2].

Other registers altered:

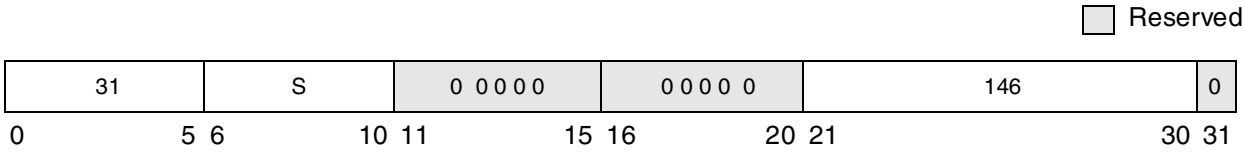
- Condition register (CR1 field):
  Affected: FX, FEX, VX, OX (if Rc = 1)
- Floating-point status and control register:
  Affected: FPSCR field **crf**D

Architecture Level	Supervisor Level	Optional	Form
UISA			X

mtmsr mtmsr

Move to Machine State Register

mtmsr **r**S




```
MSR ← (rS)
```

The contents of **r**S are placed into the MSR.

This is a supervisor-level instruction. It is also an execution synchronizing instruction except with respect to alterations to the POW and LE bits. Refer to Section 2.3.17, “Synchronization Requirements for Special Registers and for Lookaside Buffers,” for more information.

In addition, alterations to the MSR[EE] and MSR[RI] bits are effective as soon as the instruction completes. Thus if MSR[EE] = 0 and an external or decrementer interrupt is pending, executing an **mtmsr** instruction that sets MSR[EE] = 1 will cause the external or decrementer interrupt to be taken before the next instruction is executed, if no higher priority interrupt exists.

Other registers altered:

- MSR

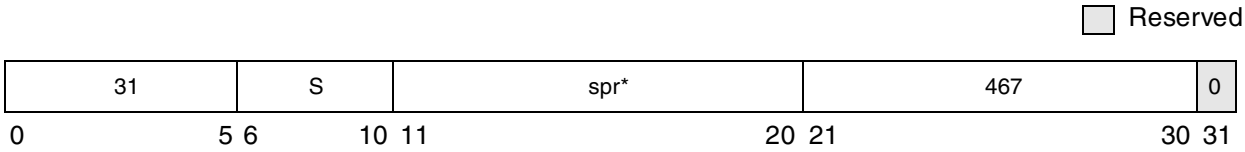
Architecture Level	Supervisor Level	Optional	Form
OEA	√		X

mtspr

mtspr

Move to Special-Purpose Register

mtspr SPR**,rS**



***Note:** This is a split field.

```
n ← spr[5:9] || spr[0:4]
  SPR(n) ← (rS)
```

In the UISA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-15. The contents of **r**S are placed into the designated special-purpose register.

Table 8-15. UISA SPR Encodings for mtspr

SPR**			Register Name
Decimal	spr[5–9]	spr[0–4]	
1	00000	00001	XER
8	00000	01000	LR
9	00000	01001	CTR

** Note that the order of the two 5-bit halves of the SPR number is reversed compared with actual instruction coding.

If the SPR field contains any value other than one of the values shown in Table 8-15, and the processor is operating in user mode, one of the following occurs:

- The system illegal instruction error handler is invoked.
- The system supervisor instruction error handler is invoked.
- The results are boundedly undefined.

Other registers altered:

- See Table 8-15.

Simplified mnemonics:

mtxer	**r**D	equivalent to	**mtspr 1,r**D
mtlr	**r**D	equivalent to	**mtspr 8,r**D
mtctr	**r**D	equivalent to	**mtspr 9,r**D

In the OEA, the SPR field denotes a special-purpose register, encoded as shown in Table 8-16. The contents of **r**S are placed into the designated special-purpose register. -

For this instruction, SPRs TBL and TBU are treated as separate 32-bit registers; setting one leaves the other unaltered.

The value of SPR[0] = 1 if and only if writing the register is a supervisor-level operation. Execution of this instruction specifying a defined and supervisor-level register when MSR[PR] = 1 results in a privileged instruction type program interrupt.

If MSR[PR] = 1 then the only effect of executing an instruction with an SPR number that is not shown in Table 8-16 and has SPR[0] = 1 is to cause a privileged instruction type program interrupt or an illegal instruction type program interrupt. For all other cases, MSR[PR] = 0 or SPR[0] = 0, if the SPR field contains any value that is not shown in Table 8-16, either an illegal instruction type program interrupt occurs or the results are boundedly undefined.

Other registers altered:

- See Table 8-16

Table 8-16. OEA SPR Encodings for mtspr

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
1	00000	00001	XER	User
8	00000	01000	LR	User
9	00000	01001	CTR	User
18	00000	10010	DSISR	Supervisor
19	00000	10011	DAR	Supervisor
22	00000	10110	DEC	Supervisor
25	00000	11001	SDR1	Supervisor
26	00000	11010	SRR0	Supervisor
27	00000	11011	SRR1	Supervisor
272	01000	10000	SPRG0	Supervisor
273	01000	10001	SPRG1	Supervisor
274	01000	10010	SPRG2	Supervisor
275	01000	10011	SPRG3	Supervisor
282	01000	11010	EAR	Supervisor

Table 8-16. OEA SPR Encodings for mtspr (continued)

SPR[1]			Register Name	Access
Decimal	spr[5–9]	spr[0–4]		
284	01000	11100	TBL	Supervisor
285	01000	11101	TBU	Supervisor
528	10000	10000	IBAT0U	Supervisor
529	10000	10001	IBAT0L	Supervisor
530	10000	10010	IBAT1U	Supervisor
531	10000	10011	IBAT1L	Supervisor
532	10000	10100	IBAT2U	Supervisor
533	10000	10101	IBAT2L	Supervisor
534	10000	10110	IBAT3U	Supervisor
535	10000	10111	IBAT3L	Supervisor
536	10000	11000	DBAT0U	Supervisor
537	10000	11001	DBAT0L	Supervisor
538	10000	11010	DBAT1U	Supervisor
539	10000	11011	DBAT1L	Supervisor
540	10000	11100	DBAT2U	Supervisor
541	10000	11101	DBAT2L	Supervisor
542	10000	11110	DBAT3U	Supervisor
543	10000	11111	DBAT3L	Supervisor
1013	11111	10101	DABR	Supervisor

[1]Note that the order of the two 5-bit halves of the SPR number is reversed. For **mtspr** and **mfspr** instructions, the SPR number coded in assembly language does not appear directly as a 10-bit binary number in the instruction. The number coded is split into two 5-bit halves that are reversed in the instruction, with the high-order five bits appearing in bits 16–20 of the instruction and the low-order five bits in bits 11–15.

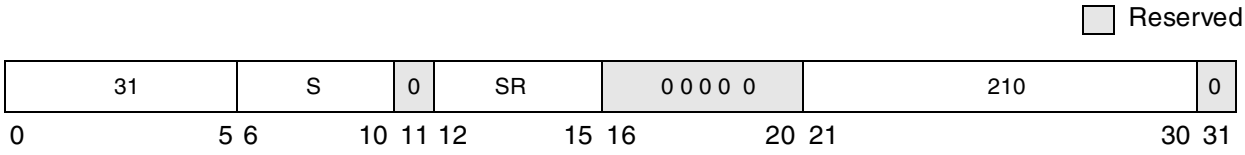
Architecture Level	Supervisor Level	Optional	Form
UISA/OEA	√*		XFX

* Note that **mtspr** is supervisor-level only if SPR[0] = 1.

mtsr mtsr

Move to Segment Register

mtsr SR,**r**S



SEGREG(SR) ← (**r**S)

The contents of **r**S are placed into SR.

This is a supervisor-level instruction.

Other registers altered:

- None

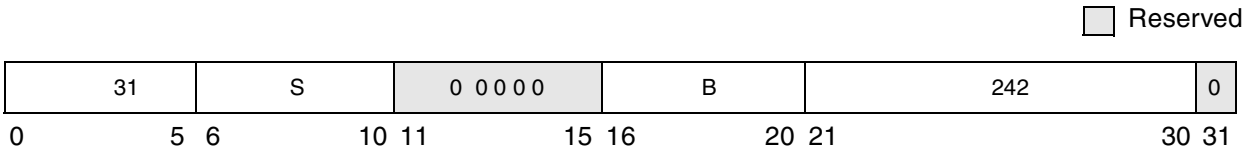
Architecture Level	Supervisor Level	Optional	Form
OEA	√		X

mtsrin

mtsrin

Move to Segment Register Indirect

mtsrin **r**S,**r**B




```
SEGREG(rB[0:3]) ← (rS)
```

The contents of **r**S are copied to the segment register selected by bits 0–3 of **r**B.

This is a supervisor-level instruction.

Note that the architecture does not define the **r**A field for the **mtsrin** instruction.

Other registers altered:

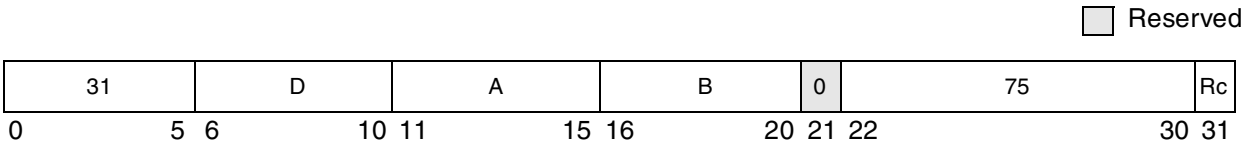
- None

Architecture Level	Supervisor Level	Optional	Form
OEA	√		X

mulhw*x* mulhw*x*

Multiply High Word

mulhw **r**D,**r**A,**r**B (Rc = 0)
mulhw. **r**D,**r**A,**r**B (Rc = 1)




```
prod[0:63] ← rA * rB
rD ← prod
```

Both the operands and the product are interpreted as signed integers.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

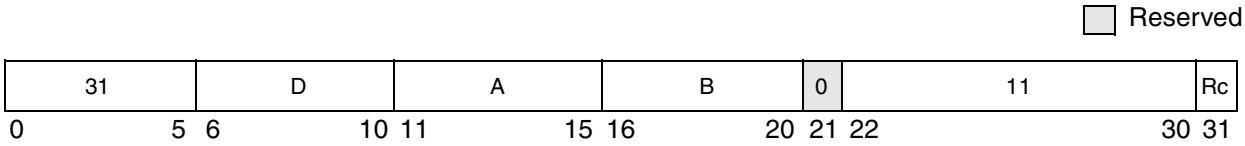
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 32-bit result.

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

mulhwu*x* mulhwu*x*

Multiply High Word Unsigned

mulhwu **r**D,**r**A,**r**B (Rc = 0)
mulhwu. **r**D,**r**A,**r**B (Rc = 1)




```
prod[0:63] ← rA * rB
rD ← prod[0:31]
```

Both the operands and the product are interpreted as unsigned integers, except that if Rc = 1 the first three bits of CR0 field are set by signed comparison of the result to zero.

This instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

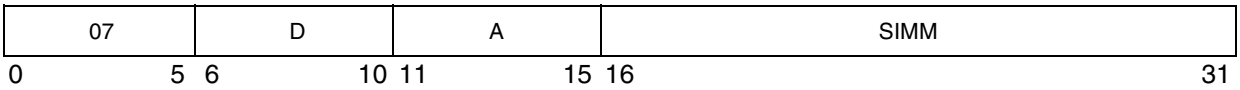
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: The setting of CR0 bits LT, GT, and EQ is mode-dependent, and reflects overflow of the 32-bit result.

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

mulli mulli

Multiply Low Immediate

mulli **r**D,**r**A,SIMM




```
prod[0:48] ← (rA) * SIMM
rD ← prod[16:48]
```

The 32-bit first operand is (**r**A). The 16-bit second operand is the value of the SIMM field. The low-order 32-bits of the 48-bit product of the operands are placed into **r**D.

Both the operands and the product are interpreted as signed integers. The low-order 32 bits of the product are calculated independently of whether the operands are treated as signed or unsigned 32-bit integers.

This instruction can be used with **mulhd***x* or **mulhw***x* to calculate a full 64-bit product.

The low-order 32 bits of the product are the correct 32-bit product for 32-bit implementations.

Other registers altered:

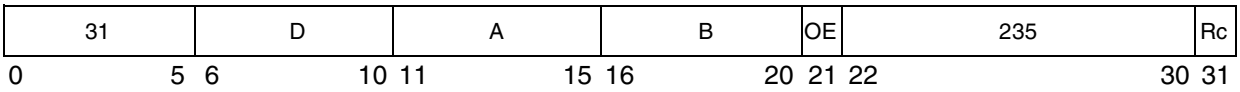
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

mullw*x* mullw*x*

Multiply Low Word

mullw	**r**D,**r**A,**r**B	(OE = 0 Rc = 0)
mullw.	**r**D,**r**A,**r**B	(OE = 0 Rc = 1)
mullwo	**r**D,**r**A,**r**B	(OE = 1 Rc = 0)
mullwo.	**r**D,**r**A,**r**B	(OE = 1 Rc = 1)




```
rD ← rA * rB
```

The 32-bit operands are the contents of **r**A and **r**B. The low-order 32 bits of the 64-bit product (**r**A) * (**r**B) are placed into **r**D.

The low-order 32 bits of the product are the correct 32-bit product for 32-bit implementations. The low-order 32-bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

If OE = 1, then OV is set if the product cannot be represented in 32 bits. Both the operands and the product are interpreted as signed integers.

Note that this instruction may execute faster on some implementations if **r**B contains the operand having the smaller absolute value.

Other registers altered:

- Condition register (CR0 field):

  Affected: LT, GT, EQ, SO (if Rc = 1)

  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:

  Affected: SO, OV (if OE = 1)

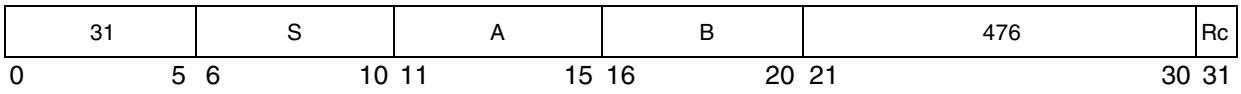
  Note: The setting of the affected bits in the XER is mode-independent, and reflects overflow of the 32-bit result.

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

nand*x* nand*x*

NAND

nand **r**A,**r**S,**r**B (Rc = 0)
nand. **r**A,**r**S,**r**B (Rc = 1)




```
rA ← ¬ ((rS) & (rB))
```

The contents of **r**S are ANDed with the contents of **r**B and the complemented result is placed into **r**A.

nand with **r**S = **r**B can be used to obtain the one's complement.

Other registers altered:

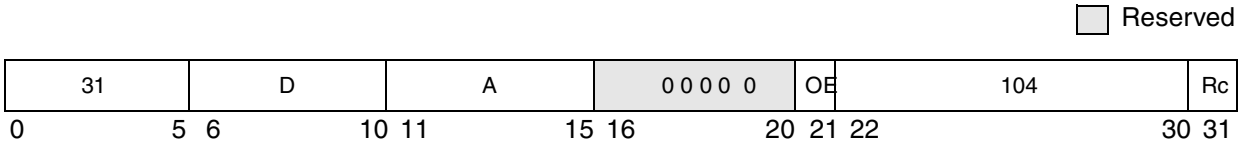
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

negx negx

Negate

neg	**r**D,**r**A	(OE = 0 Rc = 0)
neg.	**r**D,**r**A	(OE = 0 Rc = 1)
nego	**r**D,**r**A	(OE = 1 Rc = 0)
nego.	**r**D,**r**A	(OE = 1 Rc = 1)




```
rD ← ¬ (rA) + 1
```

The value 1 is added to the complement of the value in **r**A, and the resulting two's complement is placed into **r**D.

If **r**A contains the most negative 32-bit number (0x8000_0000), the result is the most negative number and, if OE = 1, OV is set.

Other registers altered:

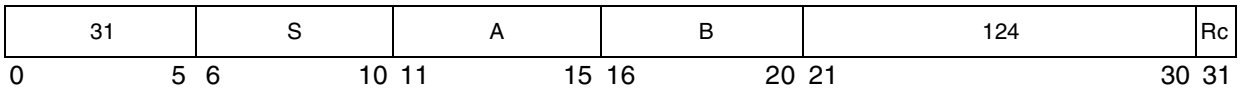
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: SO OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

norx norx

NOR

nor **r**A,**r**S,**r**B (Rc = 0)
nor. **r**A,**r**S,**r**B (Rc = 1)




```
rA ← ¬ ((rS) | (rB))
```

The contents of **r**S are ORed with the contents of **r**B and the complemented result is placed into **r**A.

nor with **r**S = **r**B can be used to obtain the one's complement.

Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Simplified mnemonics:

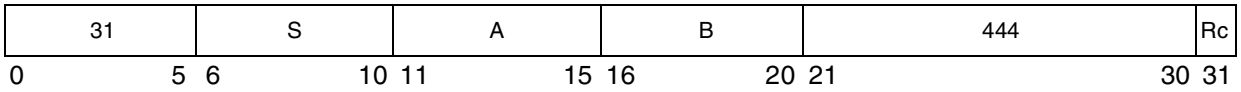
not **r**D,**r**S equivalent to **nor** **r**A,**r**S,**r**S

Architecture Level	Supervisor Level	Optional	Form
UISA			X

orx orx

OR

or **r**A,**r**S,**r**B (Rc = 0)
or. **r**A,**r**S,**r**B (Rc = 1)




```
rA ← (rS) | (rB)
```

The contents of **r**S are ORed with the contents of **r**B and the result is placed into **r**A.

The simplified mnemonic **mr** (shown below) demonstrates the use of the **or** instruction to move register contents.

Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Simplified mnemonics:

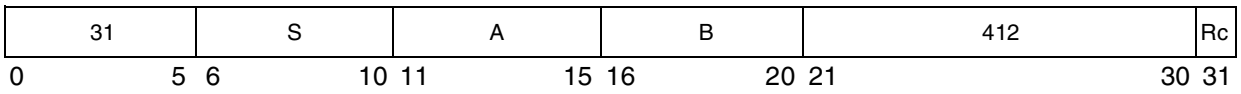
mr **r**A,**r**S equivalent to **or** **r**A,**r**S,**r**S

Architecture Level	Supervisor Level	Optional	Form
UISA			X

orc*x* orc*x*

OR with Complement

orc **r**A,**r**S,**r**B (Rc = 0)
orc. **r**A,**r**S,**r**B (Rc = 1)




```
rA ← (rS) | ¬ (rB)
```

The contents of **r**S are ORed with the complement of the contents of **r**B and the result is placed into **r**A.

Other registers altered:

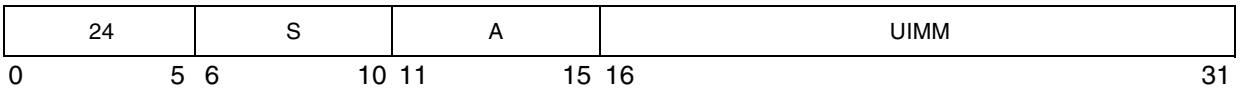
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

ori ori

OR Immediate

ori **r**A,**r**S,UIMM




```
rA ← (rS) | ((16)0 || UIMM)
```

The contents of **r**S are ORed with 0x0000 || UIMM and the result is placed into **r**A.

The preferred no-op (an instruction that does nothing) is **ori 0,0,0**.

Other registers altered:

- None

Simplified mnemonics:

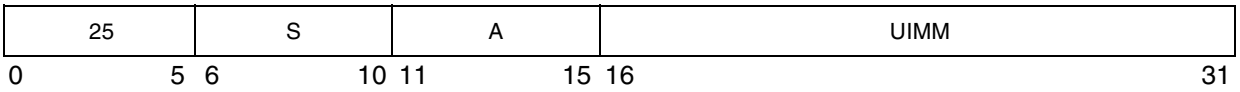
nop equivalent to **ori 0,0,0**

Architecture Level	Supervisor Level	Optional	Form
UISA			D

oris oris

OR Immediate Shifted

oris **r**A,**r**S,UIMM




```
rA ← (rS) | (UIMM || (16)0)
```

The contents of **r**S are ORed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

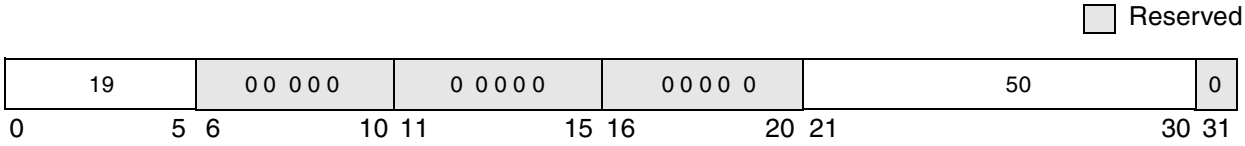
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

rfi

rfi

Return from Interrupt




```
MSR[16:23, 25:27, 30:31] ← SRR1[16:23, 25:27, 30:31]
NIA ←iea SRR0[0:29] || 0b00
```

Bits SRR1[16–23, 25–27, 30–31] are placed into the corresponding bits of the MSR. If the new MSR value does not enable any pending interrupts, then the next instruction is fetched, under control of the new MSR value, from the address SRR0[0–29] || 0b00. If the new MSR value enables one or more pending interrupts, the interrupt associated with the highest priority pending interrupt is generated; in this case the value placed into SRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not occurred. Note that an implementation may define additional MSR bits, and in this case, may also cause them to be saved to SRR1 from MSR on an interrupt and restored to MSR from SRR1 on an **rfi**.

This is a supervisor-level, context synchronizing instruction.

Other registers altered:

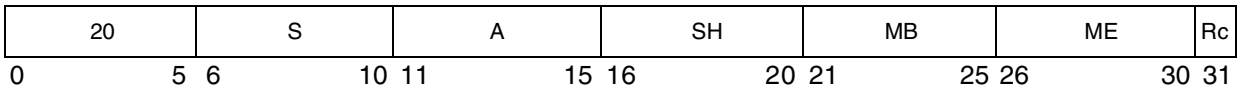
- MSR

Architecture Level	Supervisor Level	Optional	Form
OEA	√		XL

rlwimi*x* rlwimi*x*

Rotate Left Word Immediate then Mask Insert

rlwimi **r**A,**r**S,SH,MB,ME (Rc = 0)
rlwimi. **r**A,**r**S,SH,MB,ME (Rc = 1)




```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← (r & m) | (rA & ¬ m)
```

The contents of **r**S are rotated left the number of bits specified by operand SH. A mask is generated having 1 bits from bit MB through bit ME and 0 bits elsewhere. The rotated data is inserted into **r**A under control of the generated mask.

Note that **rlwimi** can be used to insert a bit field into the contents of **r**A using the methods shown below:

- To insert an *n*-bit field, that is left-justified **r**S, into **r**A starting at bit position *b*, set SH = 32 – *b*, MB = *b*, and
  ME = (*b* + *n*) – 1.
- To insert an *n*-bit field, that is right-justified in **r**S, into **r**A starting at bit position *b*, set SH = 32 – (*b* + *n*), MB = *b*, and ME = (*b* + *n*) – 1.

Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Simplified mnemonics:

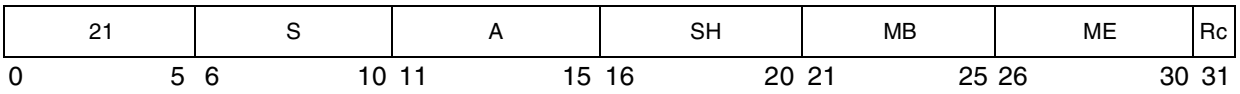
inslwi **r**A,**r**S,*n*,*b* equivalent to **rlwimi** **r**A,**r**S,32 – *b*,*b*,*b* + *n* – 1
insrwi **r**A,**r**S,*n*,*b* (n > 0) equivalent to **rlwimi** **r**A,**r**S,**32 –** (*b* + *n*),*b*,(*b* + *n*) – 1

Architecture Level	Supervisor Level	Optional	Form
UISA			M

rlwinm*x* rlwinm*x*

Rotate Left Word Immediate then AND with Mask

rlwinm **r**A,**r**S,SH,MB,ME (Rc = 0)
rlwinm. **r**A,**r**S,SH,MB,ME (Rc = 1)




```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m
```

The **r**S[0–31] contents are rotated left the number of bits specified by SH. A mask formed with 1 bits from MB through ME and 0 bits elsewhere is ANDed with the rotated data and the result is placed into **r**A. **rlwinm** can extract, rotate, shift, and clear bit fields as follows:

- To extract an *n*-bit field that starts at bit position *b* in **r**S, right-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b* + *n*, MB = 32 – *n*, and ME = 31.
- To extract an *n*-bit field that starts at bit position *b* in **r**S, left-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), set SH = *b*, MB = 0, and ME = *n* – 1.
- To rotate register contents left or right by *n* bits, set SH = *n* (32 – *n*), MB = 0, and ME = 31.
- To shift register contents right by *n* bits, set SH = 32 – *n*, MB = *n*, and ME = 31. To clear the high-order *b* bits of a register and shift the result left by *n* bits, set SH = *n*, MB = *b* – *n* and ME = 31 – *n*.
- To clear a register's low-order *n* bits , set SH = 0, MB = 0, and ME = 31 – *n*.

Other registers altered: Condition register (CR0 field): Affected: LT, GT, EQ, SO(if Rc = 1)

Simplified mnemonics:

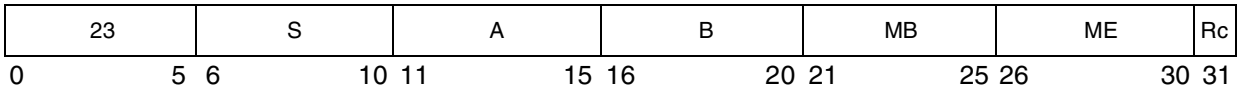
extlwi **r**A,**r**S,*n*,*b* (*n* > 0)	equivalent to	**rlwinm** **r**A,**r**S,b,**0**,*n* – 1
extrwi **r**A,**r**S,*n*,*b* (*n* > 0)	equivalent to	**rlwinm** **r**A,**r**S,b + *n*,32 – *n*,**31**
rotlwi **r**A,**r**S,*n*	equivalent to	**rlwinm** **r**A,**r**S,*n*,**0,31**
rotrwi **r**A,**r**S,*n*	equivalent to	**rlwinm** **r**A,**r**S,**32** – *n*,**0,31**
slwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,*n*,**0,**31–*n*
srwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,32 **–** *n*,*n*,**31**
clrlwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,**0,***n*,**31**
clrrwi **r**A,**r**S,*n* (*n* < 32)	equivalent to	**rlwinm** **r**A,**r**S,**0,0,**31 – *n*
clrlslwi **r**A,**r**S,*b*,*n* (*n* ≤ *b* < 32)	equivalent to	**rlwinm** **r**A,**r**S,*n*,*b* – *n*,31 – *n*

Architecture Level	Supervisor Level	Optional	Form
UISA			M

rlwnm*x* rlwnm*x*

Rotate Left Word then AND with Mask

rlwnm **r**A,**r**S,**r**B,MB,ME (Rc = 0)
rlwnm. **r**A,**r**S,**r**B,MB,ME (Rc = 1)




```
n ← rB[27:31]
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m
```

The contents of **r**S are rotated left the number of bits specified by the low-order five bits of **r**B. A mask is generated having 1 bits from bit MB through bit ME and 0 bits elsewhere. The rotated data is ANDed with the generated mask and the result is placed into **r**A.

Note that **rlwnm** can be used to extract and rotate bit fields using the methods shown as follows:

- To extract an *n*-bit field, that starts at variable bit position *b* in **r**S, right-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), by setting the low-order five bits of **r**B to *b* + *n*, MB = 32 – *n*, and ME = 31.
- To extract an *n*-bit field, that starts at variable bit position *b* in **r**S, left-justified into **r**A (clearing the remaining 32 – *n* bits of **r**A), by setting the low-order five bits of **r**B to *b*, MB = 0, and ME = *n* – 1.
- To rotate the contents of a register left (or right) by *n* bits, by setting the low-order five bits of **r**B to *n* (32 – *n*), MB = 0, and ME = 31.

Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

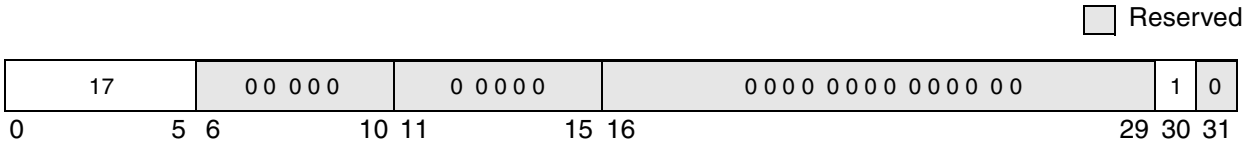
Simplified mnemonics:

rotlw **r**A,**r**S,**r**B equivalent to **rlwnm** **r**A,**r**S,**r**B,**0,31**

Architecture Level	Supervisor Level	Optional	Form
UISA			M

sc sc

System Call



In the UISA, the **sc** instruction calls the operating system to perform a service. When control is returned to the program that executed the system call, the content of the registers depends on the register conventions used by the program providing the system service.

This instruction is context synchronizing, as described in Section 4.1.4.1, “Context Synchronizing Instructions.”

Other registers altered:

- Dependent on the system service

In OEA, the **sc** instruction does the following:

```
SRR0 ←iea CIA + 4
SRR1[1:4, 10:15] ← 0
SRR1[16:23, 25:27, 30:31] ← MSR[16:23, 25:27, 30:31]
MSR ← new_value (see below)
NIA ←iea base_ea + 0xC00 (see below)
```

The EA of the instruction following the **sc** instruction is placed into SRR0. Bits 16–23, 25–27, and 30–31 of the MSR are placed into the corresponding bits of SRR1, and bits 1-4 and 10-15 of SRR1 are set to undefined values. Note that an implementation may define additional MSR bits, and in this case, may also cause them to be saved to SRR1 from MSR on an interrupt and restored to MSR from SRR1 on an **rfi**.

Then a system call interrupt is generated. The interrupt causes the MSR to be altered as described in Section 6.5, “Interrupt Definitions.”

The interrupt causes the next instruction to be fetched from offset 0xC00 from the physical base address determined by the new setting of MSR[IP].

Other registers altered:

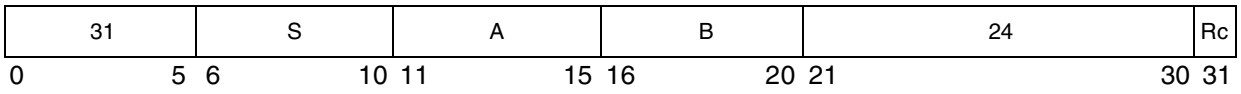
- SRR0
- SRR1
- MSR

Architecture Level	Supervisor Level	Optional	Form
UISA/OEA			SC

slwx slwx

Shift Left Word

slw **r**A,**r**S,**r**B (Rc = 0)
slw. **r**A,**r**S,**r**B (Rc = 1)




```
n ← rB[27:31]
r ← ROTL(rS, n)
```

If bit 26 of **r**B = 0, the contents of **r**S are shifted left the number of bits specified by **r**B[27–31]. Bits shifted out of position 0 are lost. Zeros are supplied to the vacated positions on the right. The 32-bit result is placed into **r**A. If bit 26 of **r**B = 1, 32 zeros are placed into **r**A.

Other registers altered:

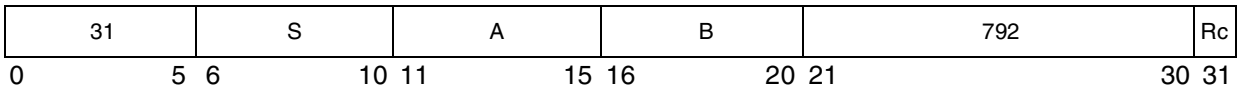
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

srawx srawx

Shift Right Algebraic Word

sraw **r**A,**r**S,**r**B (Rc = 0)
sraw. **r**A,**r**S,**r**B (Rc = 1)




```
n ← rB[27:31]
r ← ROTL(rS, n)
if rB[26] = 0 then
m ← MASK(n )
else m ← (32)0
S ← rS
rA ← r & m | S & ¬ m
XER[CA] ← S & (r & ¬ m ≠ 0
```

If **r**B[26] = 0,then the contents of **r**S are shifted right the number of bits specified by **r**B[27–31]. Bits shifted out of position 31 are lost. The result is padded on the left with sign bits before being placed into **r**A. If **r**B[26] = 1, then **r**A is filled with 32 sign bits (bit 0) from **r**S. CR0 is set based on the value written into **r**A. XER[CA] is set if **r**S contains a negative number and any 1 bits are shifted out of position 31; otherwise XER[CA] is cleared. A shift amount of zero causes XER[CA] to be cleared.

Note that the **sraw** instruction, followed by **addze**, can by used to divide quickly by 2^n.

Other registers altered:

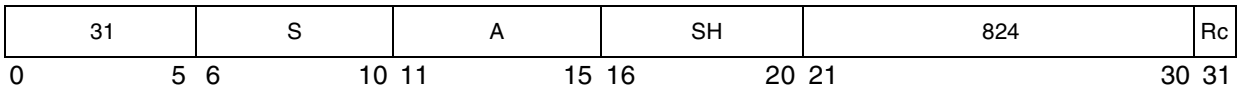
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: CA

Architecture Level	Supervisor Level	Optional	Form
UISA			X

srawi*x* srawi*x*

Shift Right Algebraic Word Immediate

srawi **r**A,**r**S,SH (Rc = 0)
srawi. **r**A,**r**S,SH (Rc = 1)




```
n ← SH
r ← ROTL(rS, 32 - n)
m← MASK(n )
S ← rS
rA ← r & m | S & ¬ m
XER[CA] ← S & ((r & ¬ m) ≠ 0)
```

The contents of **r**S are shifted right the number of bits specified by operand SH. Bits shifted out of position 31 are lost. The shifted value is sign-extended before being placed in **r**A. The 32-bit result is placed into **r**A. XER[CA] is set if **r**S contains a negative number and any 1 bits are shifted out of position 31; otherwise XER[CA] is cleared. A shift amount of zero causes XER[CA] to be cleared.

Note that the **srawi** instruction, followed by **addze**, can be used to divide quickly by 2^n.

Other registers altered:

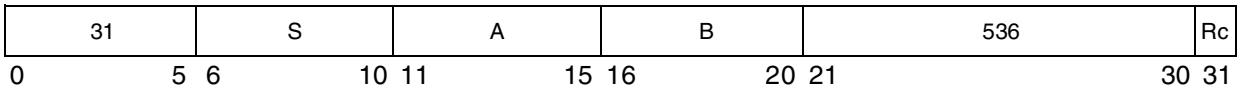
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: CA

Architecture Level	Supervisor Level	Optional	Form
UISA			X

srwx srwx

Shift Right Word

srw **r**A,**r**S,**r**B (Rc = 0)
srw. **r**A,**r**S,**r**B (Rc = 1)




```
n ← rB[27:31]
r ← ROTL(rS, 32 – n)
```

The contents of **r**S are shifted right the number of bits specified by **r**B[27–31]. Bits shifted out of position 31 are lost. Zeros are supplied to the vacated positions on the left. The result is placed into **r**A.

Other registers altered:

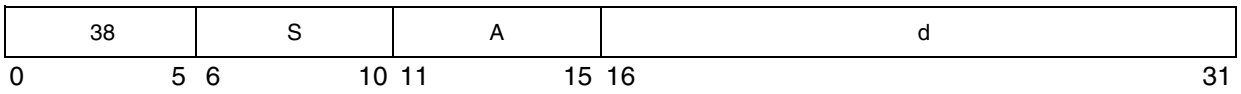
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stb stb

Store Byte

stb **r**S,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 1) ← rS[24:31]
```

EA is the sum (**r**A|0) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

Other registers altered:

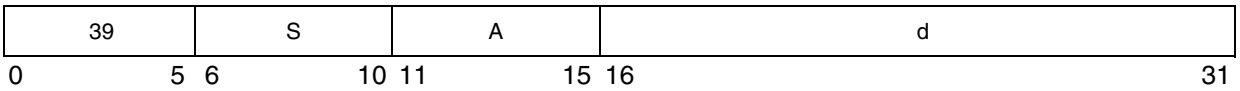
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stbu stbu

Store Byte with Update

stbu **r**S,d(**r**A)




```
EA ← (rA) + EXTS(d)
MEM(EA, 1) ← rS[24:31]
rA ← EA
```

EA is the sum (**r**A) + d. The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

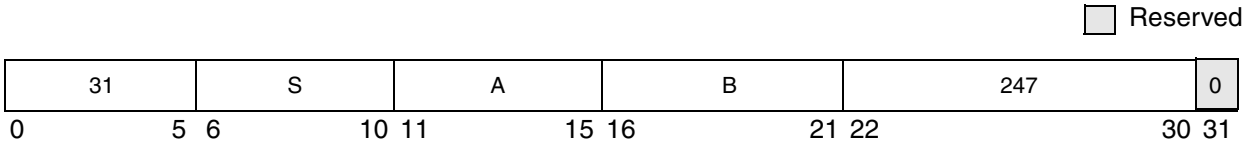
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stbux stbux

Store Byte with Update Indexed

stbux **r**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 1) ← rS[24:31]
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

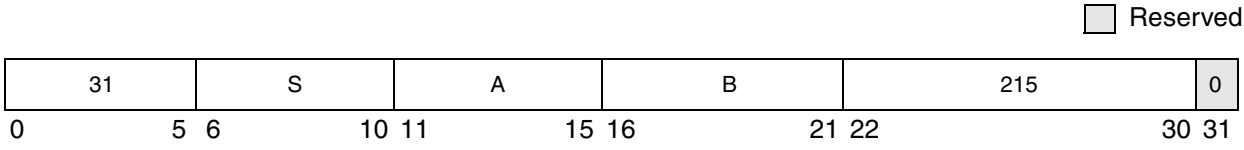
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stbx stbx

Store Byte Indexed

stbx **r**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 1) ← rS[24:31]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into the byte in memory addressed by EA.

Other registers altered:

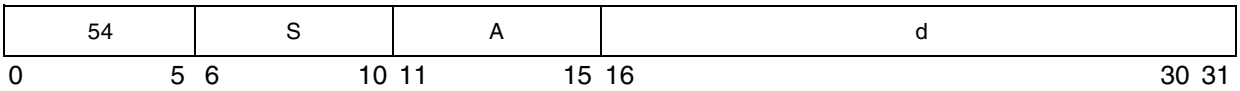
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stfd stfd

Store Floating-Point Double

stfd **fr**S,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 8) ← (frS)
```

EA is the sum (**r**A|0) + d.

The contents of register **fr**S are stored into the double word in memory addressed by EA.

Other registers altered:

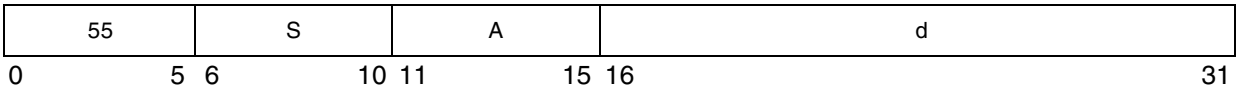
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stfdu stfdu

Store Floating-Point Double with Update

stfdu **fr**S,d(**r**A)




```
EA ← (rA) + EXTS(d)
MEM(EA, 8) ← (frS)
rA ← EA
```

EA is the sum (**r**A) + d.

The contents of register **fr**S are stored into the double word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

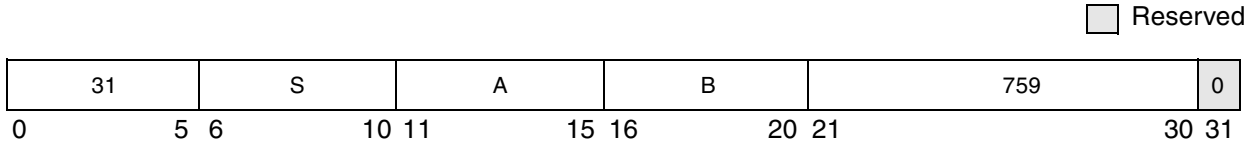
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stfdux stfdux

Store Floating-Point Double with Update Indexed

stfdux **fr**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 8) ← (frS)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The contents of register **fr**S are stored into the double word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

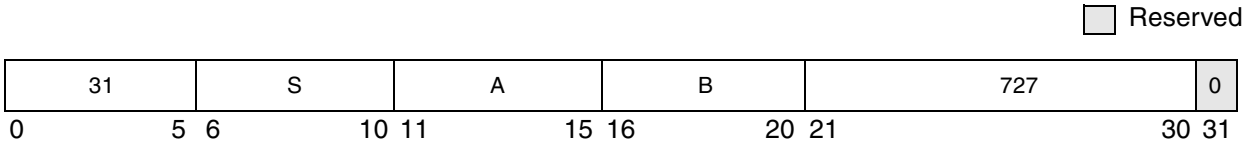
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stfdx stfdx

Store Floating-Point Double Indexed

stfdx **fr**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 8) ← (frS)
```

EA is the sum (**r**A|0) + **r**B.

The contents of register **fr**S are stored into the double word in memory addressed by EA.

Other registers altered:

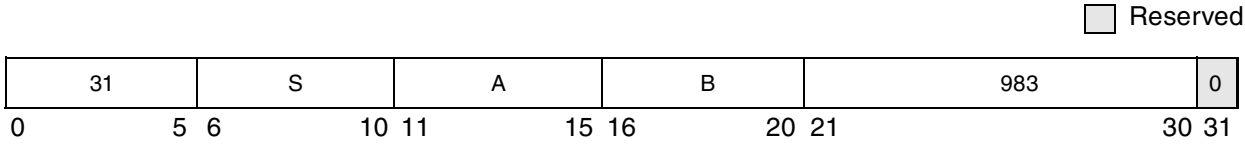
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stfiwx stfiwx

Store Floating-Point as Integer Word Indexed

stfiwx **fr**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← frS
```

EA is the sum (**r**A|0) + (**r**B).

The low-order 32 bits of **fr**S are stored, without conversion, into the word in memory addressed by EA.

If the contents of register **fr**S were produced, either directly or indirectly, by an **lfs** instruction, a single-precision arithmetic instruction, or **frsp**, then the value stored is undefined. The contents of **fr**S are produced directly by such an instruction if **fr**S is the target register for the instruction. The contents of **fr**S are produced indirectly by such an instruction if **fr**S is the final target register of a sequence of one or more floating-point move instructions, with the input to the sequence having been produced directly by such an instruction.

This instruction is defined as optional by the architecture to ensure backwards compatibility with earlier processors; however, it will likely be required for subsequent processors.

Other registers altered:

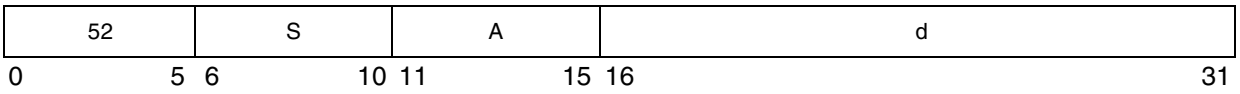
- None

Architecture Level	Supervisor Level	Optional	Form
UISA		√	X

stfs stfs

Store Floating-Point Single

stfs **fr**S,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
```

EA is the sum (**r**A|0) + d.

The contents of register **fr**S are converted to single-precision and stored into the word in memory addressed by EA. Note that the value to be stored should be in single-precision format prior to the execution of the **stfs** instruction. For a discussion on floating-point store conversions, see Section C.7, “Floating-Point Store Instructions.”

Other registers altered:

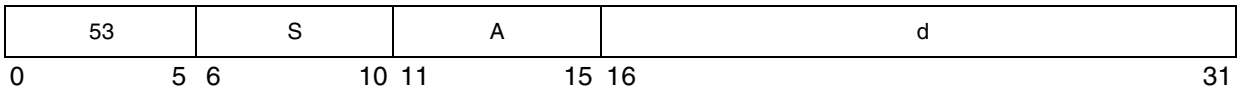
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stfsu stfsu

Store Floating-Point Single with Update

stfsu **fr**S,d(**r**A)




```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (**r**A) + d.

The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by EA. Note that the value to be stored should be in single-precision format prior to the execution of the **stfsu** instruction. For a discussion on floating-point store conversions, see Section C.7, “Floating-Point Store Instructions.”

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

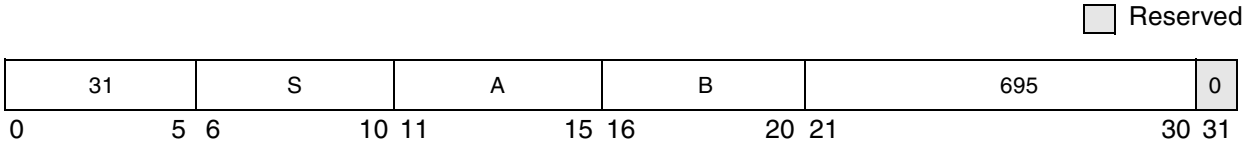
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stfsux stfsux

Store Floating-Point Single with Update Indexed

stfsux **fr**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 4) ← SINGLE(frS)
rA ← EA
```

EA is the sum (**r**A) + (**r**B).

The contents of **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see Section C.7, “Floating-Point Store Instructions.”

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

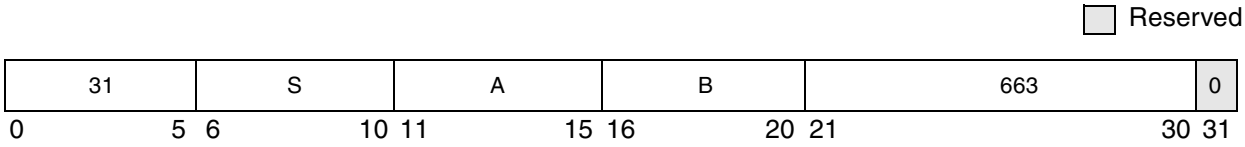
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stfsx stfsx

Store Floating-Point Single Indexed

stfsx **fr**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← SINGLE(frS)
```

EA is the sum (**r**A|0) + (**r**B).

The contents of register **fr**S are converted to single-precision and stored into the word in memory addressed by EA. For a discussion on floating-point store conversions, see Section C.7, “Floating-Point Store Instructions.”

Other registers altered:

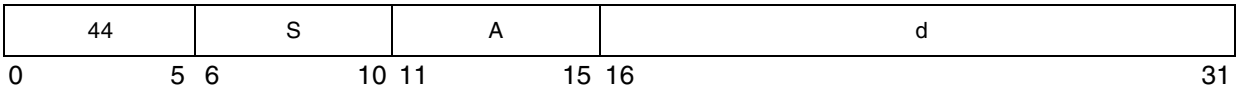
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

sth sth

Store Half Word

sth **r**S,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 2) ← rS[16:31]
```

EA is the sum (**r**A|0) + d. The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

Other registers altered:

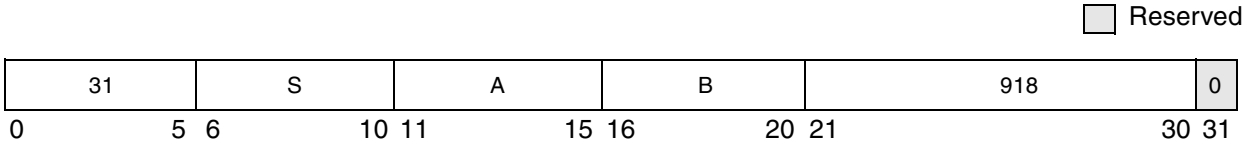
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

sthbrx sthbrx

Store Half Word Byte-Reverse Indexed

sthbrx **r**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[24:31] || rS[16:23]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into bits 0–7 of the half word in memory addressed by EA. The contents of the subsequent low-order eight bits of **r**S are stored into bits 8–15 of the half word in memory addressed by EA.

Other registers altered:

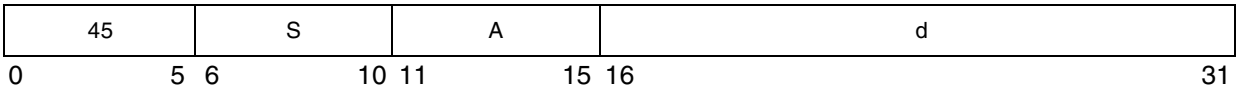
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

sthu sthu

Store Half Word with Update

sthu **r**S,d(**r**A)




```
EA ← (rA) + EXTS(d)
MEM(EA, 2) ← rS[16:31]
rA ← EA
```

EA is the sum (**r**A) + d. The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

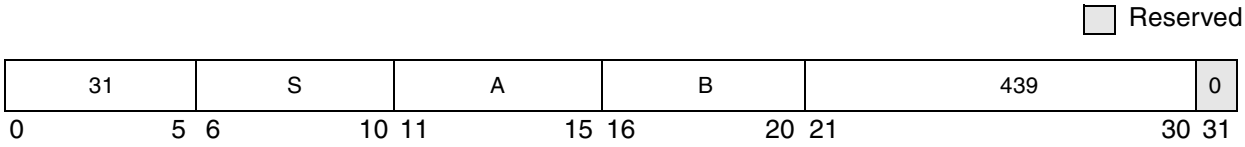
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

sthux sthux

Store Half Word with Update Indexed

sthux **r**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 2) ← rS[16:31]
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

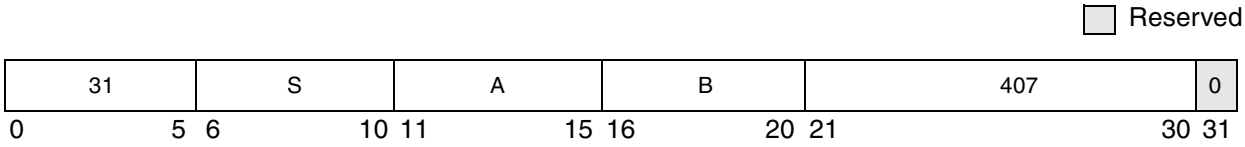
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

sthx sthx

Store Half Word Indexed

sthx **r**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 2) ← rS[16:31]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order 16 bits of **r**S are stored into the half word in memory addressed by EA.

Other registers altered:

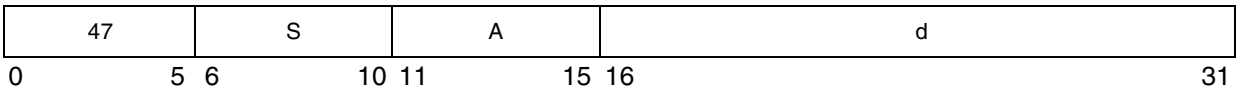
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stmw stmw

Store Multiple Word

stmw **r**S,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
r ← rS
do while r ≤ 31
   MEM(EA, 4) ← GPR(r)
   r ← r + 1
   EA ← EA + 4
```

EA is the sum (**r**A|0) + d.

n = (32 – **r**S).

n consecutive words starting at EA are stored from the GPRs **r**S through **r31**. For example, if **r**S = 30, 2 words are stored.

EA must be a multiple of four. If it is not, either the system alignment interrupt handler is invoked or the results are boundedly undefined. For additional information about alignment and DSI interrupts, see Section 6.5.3, “Data Storage Interrupt (0x00300).”

Note that, in some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

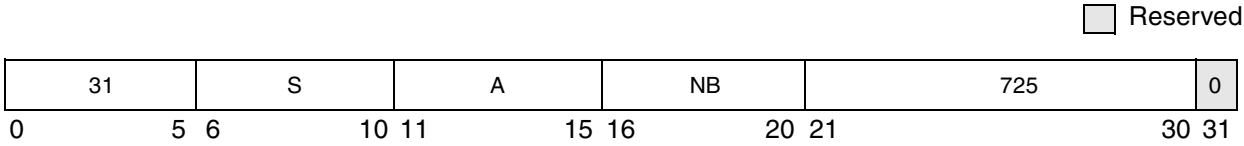
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stswi stswi

Store String Word Immediate

stswi **r**S,**r**A,NB




```
if rA = 0 then EA ← 0
else     EA ← (rA)
if NB = 0 then n ← 32
else     n ← NB
r ← rS - 1
i ← 32
do while n > 0
   if i = 32 then r ← r + 1 (mod 32)
   MEM(EA, 1) ← GPR(r)[i-i + 7]
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

EA is (**r**A|0). Let *n* = NB if NB ≠ 0, *n* = 32 if NB = 0; *n* is the number of bytes to store. Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs **r**S through **r**S + *nr* – 1. Bytes are stored left to right from each register. The sequence of registers wraps around through **r0** if required.

Under certain conditions (for example, segment boundary crossing) the data alignment interrupt handler may be invoked. For additional information about data alignment interrupts, see Section 6.5.3, “Data Storage Interrupt (0x00300).”

Note that, in some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

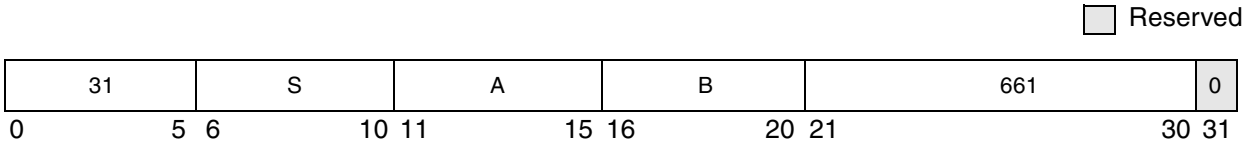
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stswx stswx

Store String Word Indexed

stswx **r**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
n ← XER[25:31]
r ← rS - 1
i ← 32
do while n > 0
   if i = 32 then r ← r + 1 (mod 32)
   MEM(EA, 1) ← GPR(r)[i-i + 7]
   i ← i + 8
   if i = 64 then i ← 32
   EA ← EA + 1
   n ← n - 1
```

EA is the sum (**r**A|0) + (**r**B). Let *n* = XER[25–31]; *n* is the number of bytes to store. Let *nr* = CEIL(*n* ÷ 4); *nr* is the number of registers to supply data.

n consecutive bytes starting at EA are stored from GPRs **r**S through **r**S + *nr* – 1. Bytes are stored left to right from each register. The sequence of registers wraps around through **r0** if required. If *n* = 0, no bytes are stored.

Under certain conditions (for example, segment boundary crossing) the data alignment interrupt handler may be invoked. For additional information about data alignment interrupts, see Section 6.5.3, “Data Storage Interrupt (0x00300).”

Note that, in some implementations, this instruction is likely to have a greater latency and take longer to execute, perhaps much longer, than a sequence of individual load or store instructions that produce the same results.

Other registers altered:

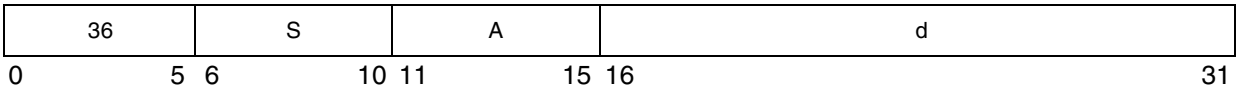
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stw stw

Store Word

stw **r**S,d(**r**A)




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4) ← rS
```

EA is the sum (**r**A|0) + d. The contents of **r**S are stored into the word in memory addressed by EA.

Other registers altered:

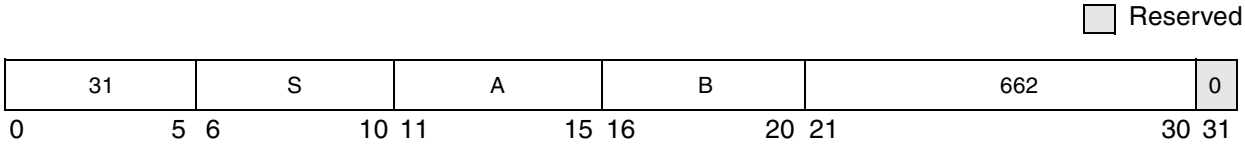
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stwbrx stwbrx

Store Word Byte-Reverse Indexed

stwbrx **r**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← rS[24:31] || rS[16:23] || rS[8:15] || rS[0:7]
```

EA is the sum (**r**A|0) + (**r**B). The contents of the low-order eight bits of **r**S are stored into bits 0–7 of the word in memory addressed by EA. The contents of the subsequent eight low-order bits of **r**S are stored into bits 8–15 of the word in memory addressed by EA. The contents of the subsequent eight low-order bits of **r**S are stored into bits 16–23 of the word in memory addressed by EA. The contents of the subsequent eight low-order bits of **r**S are stored into bits 24–31 of the word in memory addressed by EA.

Other registers altered:

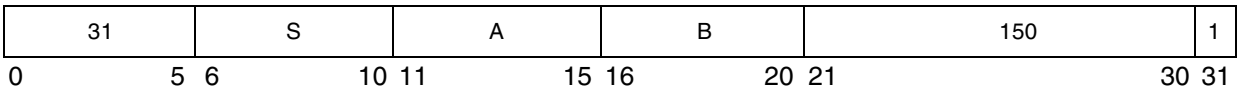
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stwcx. stwcx.

Store Word Conditional Indexed

stwcx. **r**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else      b ← (rA)
EA ← b + (rB)
if RESERVE then
  if RESERVE_ADDR = physical_addr(EA)
    MEM(EA, 4) ← rS
    CR0 ← 0b00 || 0b1 || XER[SO]
  else
    u ← undefined 1-bit value
    if u then MEM(EA, 4) ← rS
    CR0 ← 0b00 || u || XER[SO]
  RESERVE ← 0
else
  CR0 ← 0b00 || 0b0 || XER[SO]
```

EA is the sum (**r**A|0) + (**r**B). If the reserved bit is set, **stwcx.** stores **r**S to effective address (**r**A + **r**B), clears the reserved bit, and sets CR0[EQ]. If the reserved bit is not set, **stwcx.** does not do a store; it leaves the reserved bit cleared and clears CR0[EQ]. Software must look at CR0[EQ] to see if the **stwcx.** was successful.

The reserved bit is set by the **lwarx** instruction and is cleared by any **stwcx.** to any address and by snooping logic if it detects that another processor does any kind of store to the block indicated in the reservation buffer when reserved is set.

If a reservation exists and the memory address specified by the **stwcx.** is the same as that specified by the load and reserve instruction that established the reservation, the contents of **r**S are stored into the word in memory addressed by EA and the reservation is cleared.

If a reservation exists, but the address specified by the **stwcx.** is not the same as that specified by the instruction that established the reservation, the reservation is cleared and it is undefined whether the **r**S contents are stored into the word in memory addressed by EA.

If no reservation exists, the instruction completes without altering memory.

CR0 is set to reflect whether the store operation was performed as follows.

```
CR0[LT GT EQ S0] = 0b00 || store_performed || XER[SO]
```

If EA is not a multiple of four, either the system alignment interrupt handler is invoked or the results are boundedly undefined. See Section 6.5.3, “Data Storage Interrupt (0x00300).”

The granularity with which reservations are managed is implementation-dependent. Therefore, the memory to be accessed by the load and reserve and store conditional instructions should be allocated by a system library program.

Other registers altered:

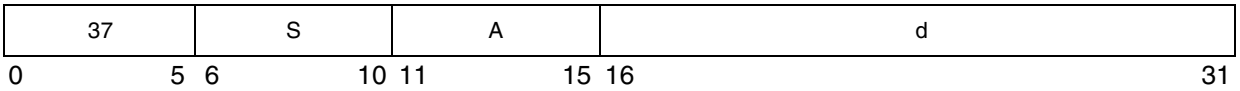
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stwu stwu

Store Word with Update

stwu **r**S,d(**r**A)




```
EA ← (rA) + EXTS(d)
MEM(EA, 4) ← rS
rA ← EA
```

EA is the sum (**r**A) + d. The contents of **r**S are stored into the word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

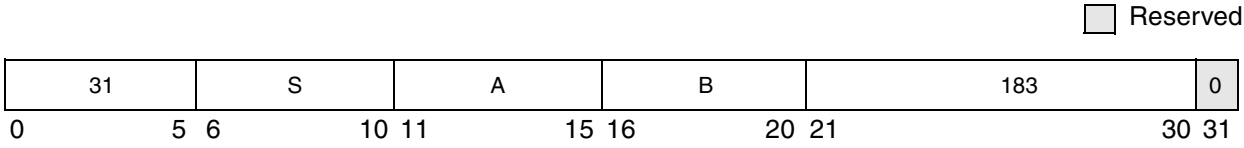
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

stwux stwux

Store Word with Update Indexed

stwux **r**S,**r**A,**r**B




```
EA ← (rA) + (rB)
MEM(EA, 4) ← rS
rA ← EA
```

EA is the sum (**r**A) + (**r**B). The contents of **r**S are stored into the word in memory addressed by EA.

EA is placed into **r**A.

If **r**A = 0, the instruction form is invalid.

Other registers altered:

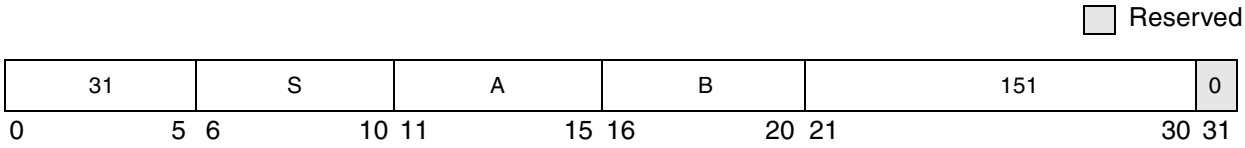
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

stwx stwx

Store Word Indexed

stwx **r**S,**r**A,**r**B




```
if rA = 0 then b ← 0
else     b ← (rA)
EA ← b + (rB)
MEM(EA, 4) ← rS
```

EA is the sum (**r**A|0) + (**r**B). The contents of **r**S are is stored into the word in memory addressed by EA.

Other registers altered:

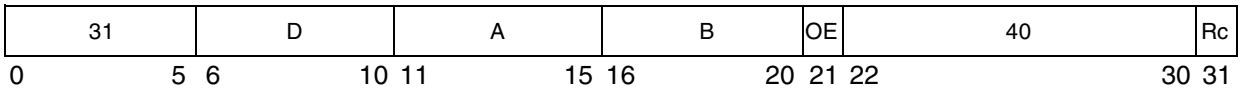
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

subf*x* subf*x*

Subtract From

subf **r**D,**r**A,**r**B (OE = 0 Rc = 0)
subf. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
subfo **r**D,**r**A,**r**B (OE = 1 Rc = 0)
subfo. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
rD ← ¬ (rA) + (rB) + 1
```

The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D.

The **subf** instruction is preferred for subtraction because it sets few status bits.

Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
- XER:
  Affected: SO, OV (if OE = 1)

Simplified mnemonics:

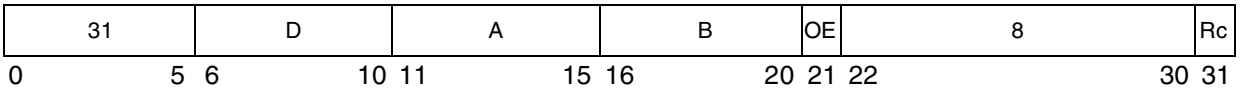
sub **r**D,**r**A,**r**B equivalent to **subf** **r**D,**r**B,**r**A

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

subfc*x* subfc*x*

Subtract from Carrying

subfc **r**D,**r**A,**r**B (OE = 0 Rc = 0)
subfc. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
subfco **r**D,**r**A,**r**B (OE = 1 Rc = 0)
subfco. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
rD ← ¬ (rA) + (rB) + 1
```

The sum ¬ (**r**A) + (**r**B) + 1 is placed into **r**D.

Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Simplified mnemonics:

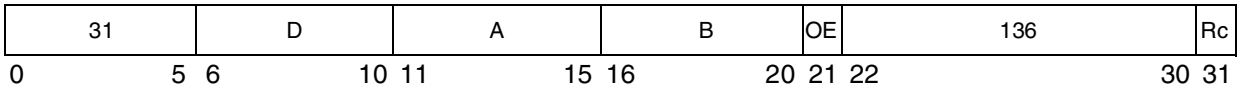
subc **r**D,**r**A,**r**B equivalent to **subfc** **r**D,**r**B,**r**A

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

subfe*x* subfe*x*

Subtract from Extended

subfe **r**D,**r**A,**r**B (OE = 0 Rc = 0)
subfe. **r**D,**r**A,**r**B (OE = 0 Rc = 1)
subfeo **r**D,**r**A,**r**B (OE = 1 Rc = 1)
subfeo. **r**D,**r**A,**r**B (OE = 1 Rc = 1)




```
rD ← ¬ (rA) + (rB) + XER[CA]
```

The sum ¬ (**r**A) + (**r**B) + XER[CA] is placed into **r**D.

Other registers altered:

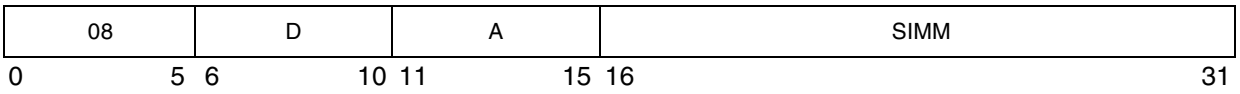
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

subfic subfic

Subtract from Immediate Carrying

subfic **r**D,**r**A,SIMM




```
rD ← ¬ (rA) + EXTS(SIMM) + 1
```

The sum ¬ (**r**A) + EXTS(SIMM) + 1 is placed into **r**D.

Other registers altered:

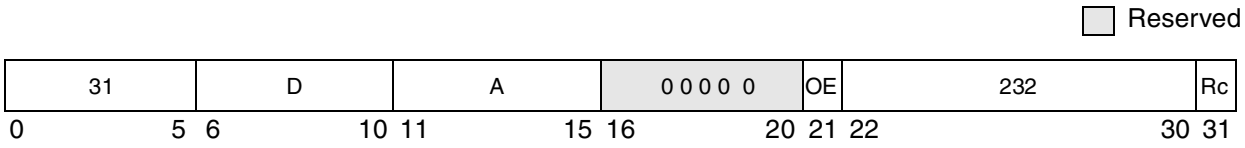
- XER:
  Affected: CA

Architecture Level	Supervisor Level	Optional	Form
UISA			D

subfme*x* subfme*x*

Subtract from Minus One Extended

subfme **r**D,**r**A (OE = 0 Rc = 0)
subfme. **r**D,**r**A (OE = 0 Rc = 1)
subfmeo **r**D,**r**A (OE = 1 Rc = 0)
subfmeo. **r**D,**r**A (OE = 1 Rc = 1)




```
rD ← ¬ (rA) + XER[CA] – 1
```

The sum ¬ (**r**A) + XER[CA] + (32)1 is placed into **r**D.

Other registers altered:

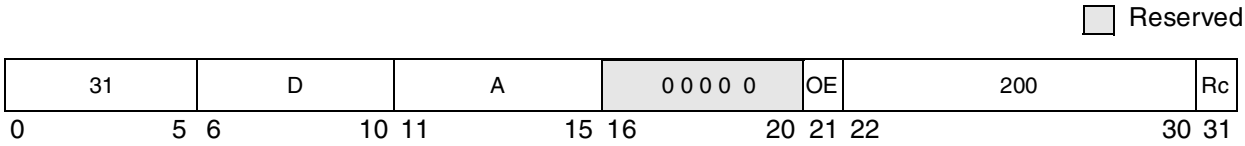
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

subfze*x* subfze*x*

Subtract from Zero Extended

subfze	**r**D,**r**A	(OE = 0 Rc = 0)
subfze.	**r**D,**r**A	(OE = 0 Rc = 1)
subfzeo	**r**D,**r**A	(OE = 1 Rc = 0)
subfzeo.	**r**D,**r**A	(OE = 1 Rc = 1)




```
rD ← ¬ (rA) + XER[CA]
```

The sum ¬ (**r**A) + XER[CA] is placed into **r**D.

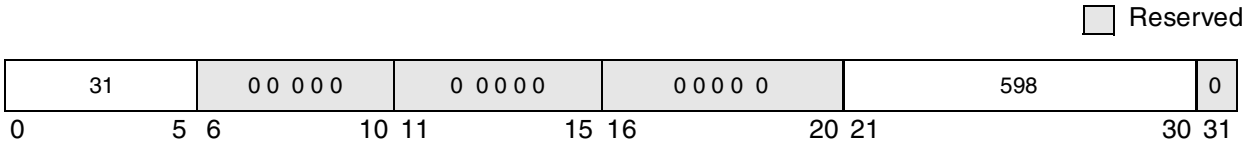
Other registers altered:

- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)
  Note: CR0 field may not reflect the infinitely precise result if overflow occurs (see XER below).
- XER:
  Affected: CA
  Affected: SO, OV (if OE = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			XO

sync sync

Synchronize



The **sync** instruction provides an ordering function for the effects of all instructions executed by a given processor. Executing a **sync** instruction ensures that all instructions preceding the **sync** instruction appear to have completed before the **sync** instruction completes, and that no subsequent instructions are initiated by the processor until after the **sync** instruction completes. When the **sync** instruction completes, all external accesses caused by instructions preceding the **sync** instruction will have been performed with respect to all other mechanisms that access memory. For more information on how the **sync** instruction affects the VEA, refer to Chapter 5, “Cache Model and Memory Coherency.”

Multiprocessor implementations also send a **sync** address-only broadcast that is useful in some designs. For example, if a design has an external buffer that re-orders loads and stores for better bus efficiency, the **sync** broadcast signals to that buffer that previous loads/stores must be completed before any following loads/stores.

The **sync** instruction can be used to ensure that the results of all stores into a data structure, caused by store instructions executed in a “critical section” of a program, are seen by other processors before the data structure is seen as unlocked.

The functions performed by the **sync** instruction will normally take a significant amount of time to complete, so indiscriminate use of this instruction may adversely affect performance. In addition, the time required to execute **sync** may vary from one execution to another.

The **eieio** instruction may be more appropriate than **sync** for many cases.

This instruction is execution synchronizing. For more information on execution synchronization, see Section 4.1.4, “Synchronizing Instructions.”

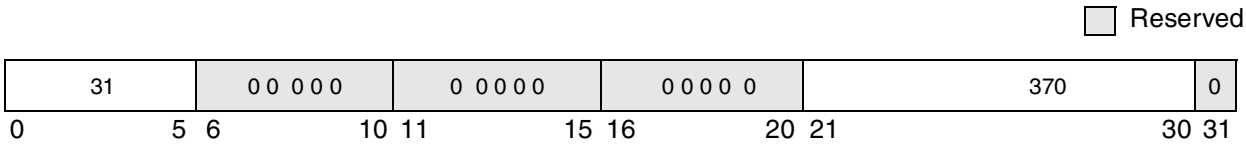
Other registers altered:

- None

Architecture Level	Supervisor Level	Optional	Form
UISA			X

tlbia tlbia

Translation Lookaside Buffer Invalidate All




```
All TLB entries ← invalid
```

The entire translation lookaside buffer (TLB) is invalidated (that is, all entries are removed).

The TLB is invalidated regardless of the settings of MSR[IR] and MSR[DR]. The invalidation is done without reference to the SLB, segment table, or segment registers.

This instruction does not cause the entries to be invalidated in other processors.

This is a supervisor-level instruction and optional in the architecture.

Other registers altered:

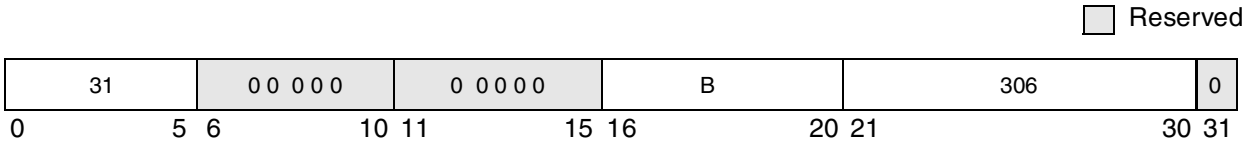
- None

Architecture Level	Supervisor Level	Optional	Form
OEA	√	√	X

tlbie tlbie

Translation Lookaside Buffer Invalidate Entry

tlbie **r**B




```
VPS ← rB[4:19]
Identify TLB entries corresponding to VPS
Each such TLB entry ← invalid
```

VPS is the page index. EA is the contents of **r**B. If the translation lookaside buffer (TLB) contains an entry corresponding to EA, that entry is made invalid (that is, removed from the TLB).

Multiprocessing implementations send a **tlbie** address-only broadcast over the address bus to tell other processors to invalidate the same TLB entry in their TLBs.

The TLB search is done regardless of the settings of MSR[IR] and MSR[DR]. The search is done based on a portion of the logical page number within a segment, without reference to the segment registers. All entries matching the search criteria are invalidated.

Block address translation for EA, if any, is ignored. Refer to Section 7.6.3.4, “Synchronization of Memory Accesses and Reference and Change Bit Updates,” and Section 7.7.3, “Page Table Updates,” for other requirements associated with the use of this instruction.

This is a supervisor-level instruction and optional in the architecture.

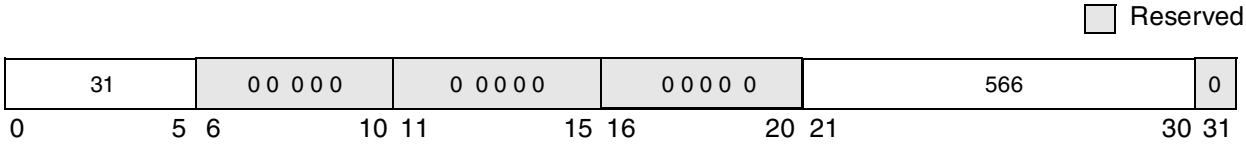
Other registers altered:

- None

Architecture Level	Supervisor Level	Optional	Form
OEA	√	√	X

tlbsync tlbsync

TLB Synchronize



If an implementation sends a broadcast for **tlbie** then it will also send a broadcast for **tlbsync**. Executing a **tlbsync** instruction ensures that all **tlbie** instructions previously executed by the processor executing the **tlbsync** instruction have completed on all other processors.

The operation performed by this instruction is treated as a caching-inhibited and guarded data access with respect to the ordering done by **eieio**.

Refer to Section 7.6.3.4, “Synchronization of Memory Accesses and Reference and Change Bit Updates,” and Section 7.7.3, “Page Table Updates,” for other requirements associated with the use of this instruction.

This instruction is supervisor-level and optional in the architecture.

Other registers altered:

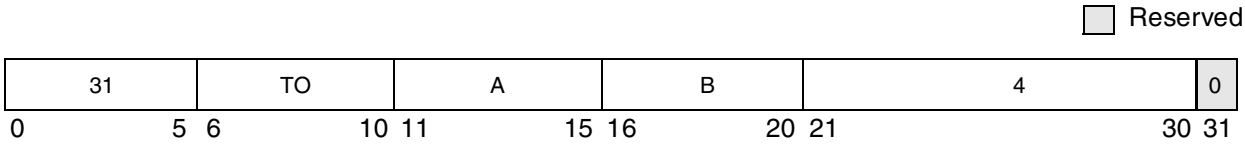
- None

Architecture Level	Supervisor Level	Optional	Form
OEA	√	√	X

tw tw

Trap Word

tw TO,**r**A,**r**B




```
a ← EXTS(rA)
b ← EXTS(rB)
if (a < b) & TO[0] then TRAP
if (a > b) & TO[1] then TRAP
if (a = b) & TO[2] then TRAP
if (a <U b) & TO[3] then TRAP
if (a >U b) & TO[4] then TRAP
```

The contents of **r**A are compared with the contents of **r**B. If any TO bit is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

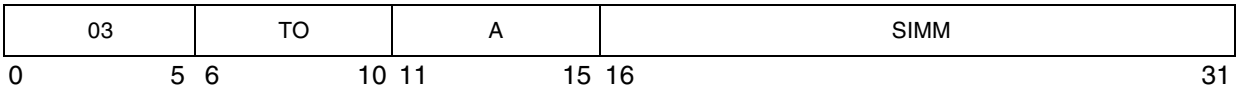
tweq	**r**A,**r**B	equivalent to	**tw**	**4,r**A,**r**B
twlge	**r**A,**r**B	equivalent to	**tw**	**5,r**A,**r**B
trap		equivalent to	**tw**	**31,0,0**

Architecture Level	Supervisor Level	Optional	Form
UISA			X

twi twi

Trap Word Immediate

twi TO,**r**A,SIMM




```
a ← EXTS(rA)
if (a < EXTS(SIMM)) & TO[0] then TRAP
if (a > EXTS(SIMM)) & TO[1] then TRAP
if (a = EXTS(SIMM)) & TO[2] then TRAP
if (a <U EXTS(SIMM)) & TO[3] then TRAP
if (a >U EXTS(SIMM)) & TO[4] then TRAP
```

The contents of **r**A are compared with the sign-extended value of the SIMM field. If any bit in the TO field is set and its corresponding condition is met by the result of the comparison, then the system trap handler is invoked.

Other registers altered:

- None

Simplified mnemonics:

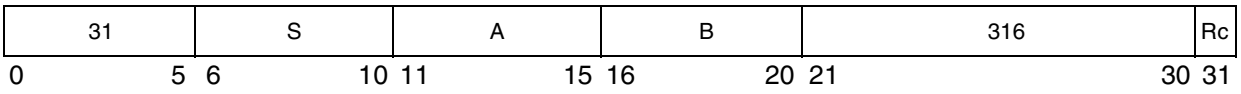
twgti	**r**A,value	equivalent to	**twi**	**8,r**A,value
twllei	**r**A,value	equivalent to	**twi**	**6,r**A,value

Architecture Level	Supervisor Level	Optional	Form
UISA			D

xor*x* xor*x*

XOR

xor **r**A,**r**S,**r**B (Rc = 0)
xor. **r**A,**r**S,**r**B (Rc = 1)



rA ← (rS) ⊕ (rB)

The contents of **r**S is XORed with the contents of **r**B and the result is placed into **r**A.

Other registers altered:

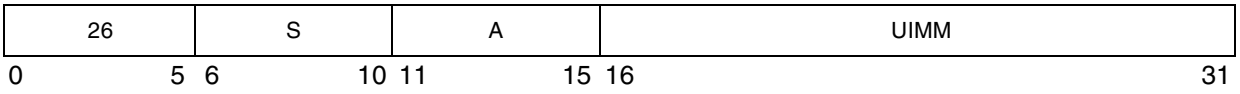
- Condition register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Architecture Level	Supervisor Level	Optional	Form
UISA			X

xori xori

XOR Immediate

xori **r**A,**r**S,UIMM




```
rA ← (rS) ⊕ ((16)0 || UIMM)
```

The contents of **r**S are XORed with 0x0000 || UIMM and the result is placed into **r**A.

Other registers altered:

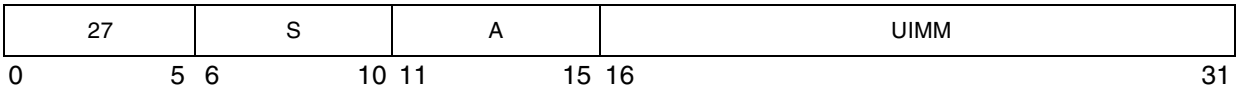
- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

xoris xoris

XOR Immediate Shifted

xoris **r**A,**r**S,UIMM




```
rA ← (rS) ⊕ (UIMM || (16)0)
```

The contents of **r**S are XORed with UIMM || 0x0000 and the result is placed into **r**A.

Other registers altered:

- None

Architecture Level	Supervisor Level	Optional	Form
UISA			D

Appendix A
Instruction Set Listings

This appendix lists the instructions by both mnemonic and opcode, and includes a quick reference table with general information, such as the architecture level, privilege level, form, and whether the instruction is optional. The tables in the chapter are organized as follows:

- Section A.1, “Instructions Sorted by Mnemonic (Decimal and Hexadecimal)”
- Section A.2, “Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal)”
- Section A.3, “Instructions Sorted by Mnemonic (Binary)”
- Section A.4, “Instructions Sorted by Opcode (Binary)”
- Section A.5, “Instruction Set Legend”

A.1 Instructions Sorted by Mnemonic (Decimal and Hexadecimal)

Table A-1 lists instructions in alphabetical order by mnemonic, showing decimal and hexadecimal values of the primary opcode (0–5) and binary values of the secondary opcode (21–31). This list also includes simplified mnemonics and their equivalents using standard mnemonics.

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
add	31 (0x1F)	rD	rA	rB		0 1 0	0 0 0 1	0 1 0 0	X	**add**
add.	31 (0x1F)	rD	rA	rB		0 1 0	0 0 0 1	0 1 0 1	X	**add.**
addc	31 (0x1F)	rD	rA	rB		0 0 0	0 0 0 1	0 1 0 0	X	**addc**
addc.	31 (0x1F)	rD	rA	rB		0 0 0	0 0 0 1	0 1 0 1	X	**addc.**
addco	31 (0x1F)	rD	rA	rB		1 0 0	0 0 0 1	0 1 0 0	X	**addco**
addco.	31 (0x1F)	rD	rA	rB		1 0 0	0 0 0 1	0 1 0 1	X	**addco.**
adde	31 (0x1F)	rD	rA	rB		0 0 1	0 0 0 1	0 1 0 0	X	**adde**
adde.	31 (0x1F)	rD	rA	rB		0 0 1	0 0 0 1	0 1 0 1	X	**adde.**
addeo	31 (0x1F)	rD	rA	rB		1 0 1	0 0 0 1	0 1 0 0	X	**addeo**
addeo.	31 (0x1F)	rD	rA	rB		1 0 1	0 0 0 1	0 1 0 1	X	**addeo.**
addi	14 (0x0E)	rD	rA	SIMM					D	**addi**
addic	12 (0x0C)	rD	rA	SIMM					D	**addic**
addic.	13 (0x0D)	rD	rA	SIMM					D	**addic.**
addis	15 (0x0F)	rD	rA	SIMM					D	**addis**
addme	31 (0x1F)	rD	rA	///		0 0 1	1 1 0 1	0 1 0 0	X	**addme**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
addme.	31 (0x1F)	rD		rA	///	0 0 1	1 1 0 1	0 1 0 1	X	**addme.**
addmeo	31 (0x1F)	rD		rA	///	1 0 1	1 1 0 1	0 1 0 0	X	**addmeo**
addmeo.	31 (0x1F)	rD		rA	///	1 0 1	1 1 0 1	0 1 0 1	X	**addmeo.**
addo	31 (0x1F)	rD		rA	rB	1 1 0	0 0 0 1	0 1 0 0	X	**addo**
addo.	31 (0x1F)	rD		rA	rB	1 1 0	0 0 0 1	0 1 0 1	X	**addo.**
addze	31 (0x1F)	rD		rA	///	0 0 1	1 0 0 1	0 1 0 0	X	**addze**
addze.	31 (0x1F)	rD		rA	///	0 0 1	1 0 0 1	0 1 0 1	X	**addze.**
addzeo	31 (0x1F)	rD		rA	///	1 0 1	1 0 0 1	0 1 0 0	X	**addzeo**
addzeo.	31 (0x1F)	rD		rA	///	1 0 1	1 0 0 1	0 1 0 1	X	**addzeo.**
and	31 (0x1F)	rS		rA	rB	0 0 0	0 0 1 1	1 0 0 0	X	**and**
and.	31 (0x1F)	rS		rA	rB	0 0 0	0 0 1 1	1 0 0 1	X	**and.**
andc	31 (0x1F)	rS		rA	rB	0 0 0	0 1 1 1	1 0 0 0	X	**andc**
andc.	31 (0x1F)	rS		rA	rB	0 0 0	0 1 1 1	1 0 0 1	X	**andc.**
andi.	28 (0x1C)	rS		rA	UIMM				D	**andi.**
andis.	29 (0x1D)	rS		rA	UIMM				D	**andis.**
b	18 (0x12)	LI						0 0	I	**b**
ba	18 (0x12)	LI						1 0	I	**ba**
bc	16 (0x10)	BO		BI	BD			0 0	B	**bc**
bca	16 (0x10)	BO		BI	BD			1 0	B	**bca**
bcctr	19 (0x13)	BO		BI	///	1 0 0	0 0 1 0	0 0 0 0	XL	**bcctr**
bcctrl	19 (0x13)	BO		BI	///	1 0 0	0 0 1 0	0 0 0 1	XL	**bcctrl**
bcl	16 (0x10)	BO		BI	BD			0 1	B	**bcl**
bcla	16 (0x10)	BO		BI	BD			1 1	B	**bcla**
bclr	19 (0x13)	BO		BI	///	0 0 0	0 0 1 0	0 0 0 0	XL	**bclr**
bclrl	19 (0x13)	BO		BI	///	0 0 0	0 0 1 0	0 0 0 1	XL	**bclrl**

bctr	**bctr**[1]	equivalent to	**bcctr 20,0**	**bctr**
bctrl	**bctrl** [1]	equivalent to	**bcctrl 20,0**	**bctrl**
bdnz	**bdnz target** [1]	equivalent to	**bc 16,0,**target	**bdnz**
bdnza	**bdnza** target [1]	equivalent to	**bca 16,0,**target	**bdnza**
bdnzf	**bdnzf BI,**target	equivalent to	**bc 0,BI,**target	**bdnzf**
bdnzfa	**bdnzfa BI,**target	equivalent to	**bca 0,BI,**target	**bdnzfa**
bdnzfl	**bdnzfl BI,**target	equivalent to	**bcl 0,BI,**target	**bdnzfl**
bdnzfla	**bdnzfla BI,**target	equivalent to	**bcla 0,BI,**target	**bdnzfla**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
bdnzflr	**bdnzflr BI**		equivalent to		**bclr 0,BI**					**bdnzflr**
bdnzflrl	**bdnzflrl BI**		equivalent to		**bclrl 0,BI**					**bdnzflrl**
bdnzl	**bdnzl** target [1]		equivalent to		**bcl 16,0,**target					**bdnzl**
bdnzla	**bdnzla** target [1]		equivalent to		**bcla 16,0,**target					**bdnzla**
bdnzlr	**bdnzlr BI**		equivalent to		**bclr 16,BI**					**bdnzlr**
bdnzlrl	**bdnzlrl** [1]		equivalent to		**bclrl 16,0**					**bdnzlrl**
bdnzt	**bdnzt BI,**target		equivalent to		**bc 8,BI,**target					**bdnzt**
bdnzta	**bdnzta BI,**target		equivalent to		**bca 8,BI,**target					**bdnzta**
bdnztl	**bdnztl BI,**target		equivalent to		**bcl 8,0,**target					**bdnztl**
bdnztla	**bdnztla BI,**target		equivalent to		**bcla 8,BI,**target					**bdnztla**
bdnztlr	**bdnztlr BI**		equivalent to		**bclr 8,BI**					**bdnztlr**
bdnztlr	**bdnztlr BI**		equivalent to		**bclr 8,BI**					**bdnztlr**
bdnztlrl	**bdnztlrl BI**		equivalent to		**bclrl 8,BI**					**bdnztlrl**
bdz	**bdz** target [1]		equivalent to		**bc 18,0,**target					**bdz**
bdza	**bdza** target [1]		equivalent to		**bca 18,0,**target					**bdza**
bdzf	**bdzf BI,**target		equivalent to		**bc 2,BI,**target					**bdzf**
bdzfa	**bdzfa BI,**target		equivalent to		**bca 2,BI,**target					**bdzfa**
bdzfl	**bdzfl BI,**target		equivalent to		**bcl 2,BI,**target					**bdzfl**
bdzfla	**bdzfla BI,**target		equivalent to		**bcla 2,BI,**target					**bdzfla**
bdzflr	**bdzflr BI**		equivalent to		**bclr 2,BI**					**bdzflr**
bdzflrl	**bdzflrl BI**		equivalent to		**bclrl 2,BI**					**bdzflrl**
bdzl	**bdzl** target [1]		equivalent to		**bcl 18,BI,**target					**bdzl**
bdzla	**bdzla** target [1]		equivalent to		**bcla 18,BI,**target					**bdzla**
bdzlr	**bdzlr** [1]		equivalent to		**bclr 18,0**					**bdzlr**
bdzlrl	**bdzlrl** [1]		equivalent to		**bclrl 18,0**					**bdzlrl**
bdzt	**bdzt BI,**target		equivalent to		**bc 10,BI,**target					**bdzt**
bdzta	**bdzta BI,**target		equivalent to		**bca 10,BI,**target					**bdzta**
bdztl	**bdztl BI,**target		equivalent to		**bcl 10,BI,**target					**bdztl**
bdztla	**bdztla BI,**target		equivalent to		**bcla 10,BI,**target					**bdztla**
bdztlrl	**bdztlrl BI**		equivalent to		**bclrl 10, BI**					**bdztlrl**
beq	**beq cr**S,target		equivalent to		**bc 12,**BI[2]**,**target					**beq**
beqa	**beqa cr**S,target		equivalent to		**bca 12,**BI[2]**,**target					**beqa**
beqctr	**beqctr cr**S,target		equivalent to		**bcctr 12,**BI[2]**,**target					**beqctr**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
beqctrl	**beqctrl crS**,target		equivalent to	**bcctrl 12,**BI[2],target						**beqctrl**
beql	**beql crS**,target		equivalent to	**bcl 12,**BI[2],target						**beql**
beqla	**beqla crS**,target		equivalent to	**bcla 12,**BI[2],target						**beqla**
beqlr	**beqlr crS**,target		equivalent to	**bclr 12,**BI[2],target						**beqlr**
beqlrl	**beqlrl crS**,target		equivalent to	**bclrl 12,**BI[2],target						**beqlrl**
bf	**bf BI,**target		equivalent to	**bc 4,BI,**target						**bf**
bfa	**bfa BI,**target		equivalent to	**bca 4,BI,**target						**bfa**
bfctr	**bfctr BI**		equivalent to	**bcctr 4,BI**						**bfctr**
bfctrl	**bfctrl BI**		equivalent to	**bcctrl 4,BI**						**bfctrl**
bfl	**bfl BI,**target		equivalent to	**bcl 4,BI,**target						**bfl**
bfla	**bfla BI,**target		equivalent to	**bcla 4,BI,**target						**bfla**
bflr	**bflr BI**		equivalent to	**bclr 4,BI**						**bflr**
bflrl	**bflrl BI**		equivalent to	**bclrl 4,BI**						**bflrl**
bge	**bge crS**,target		equivalent to	**bc 4,**BI[3],target						**bge**
bgea	**bgea crS**,target		equivalent to	**bca 4,**BI[3],target						**bgea**
bgectr	**bgectr crS**,target		equivalent to	**bcctr 4,**BI[3],target						**bgectr**
bgectrl	**bgectrl crS**,target		equivalent to	**bcctrl 4,**BI[3],target						**bgectrl**
bgel	**bgel crS**,target		equivalent to	**bcl 4,**BI[3],target						**bgel**
bgela	**bgela crS**,target		equivalent to	**bcla 4,**BI[3],target						**bgela**
bgelr	**bgelr crS**,target		equivalent to	**bclr 4,**BI[3],target						**bgelr**
bgelrl	**bgelrl crS**,target		equivalent to	**bclrl 4,**BI[3],target						**bgelrl**
bgt	**bgt crS**,target		equivalent to	**bc 12,**BI[4],target						**bgt**
bgta	**bgta crS**,target		equivalent to	**bca 12,**BI[4],target						**bgta**
bgtctr	**bgtctr crS**,target		equivalent to	**bcctr 12,**BI[4],target						**bgtctr**
bgtctrl	**bgtctrl crS**,target		equivalent to	**bcctrl 12,**BI[4],target						**bgtctrl**
bgtl	**bgtl crS**,target		equivalent to	**bcl 12,**BI[4],target						**bgtl**
bgtla	**bgtla crS**,target		equivalent to	**bcla 12,**BI[4],target						**bgtla**
bgtlr	**bgtlr crS**,target		equivalent to	**bclr 12,**BI[4],target						**bgtlr**
bgtlrl	**bgtlrl crS**,target		equivalent to	**bclrl 12,**BI[4],target						**bgtlrl**
bl	18 (0x12)		LI					0 1	I	**bl**
bla	18 (0x12)		LI					1 1	I	**bla**
ble	**ble crS**,target		equivalent to	**bc 4,**BI[4],target						**ble**
blea	**blea crS**,target		equivalent to	**bca 4,**BI[4],target						**blea**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
blectr	**blectr** crS,target		equivalent to	**bcctr 4,**BI[4],target						**blectr**
blectrl	**blectrl** crS,target		equivalent to	**bcctrl 4,**BI[4],target						**blectrl**
blel	**blel** crS,target		equivalent to	**bcl 4,**BI[4],target						**blel**
blela	**blela** crS,target		equivalent to	**bcla 4,**BI[4],target						**blela**
blelr	**blelr** crS,target		equivalent to	**bclr 4,**BI[4],target						**blelr**
blelrl	**blelrl** crS,target		equivalent to	**bclrl 4,**BI[4],target						**blelrl**
blr	**blr** [1]		equivalent to	**bclr 20,0**						**blr**
blrl	**blrl** [1]		equivalent to	**bclrl 20,0**						**blrl**
blt	**blt** crS,target		equivalent to	**bc 12,**BI,target						**blt**
blta	**blta** crS,target		equivalent to	**bca 12,**BI[3],target						**blta**
bltctr	**bltctr** crS,target		equivalent to	**bcctr 12,**BI[3],target						**bltctr**
bltctrl	**bltctrl** crS,target		equivalent to	**bcctrl 12,**BI[3],target						**bltctrl**
bltl	**bltl** crS,target		equivalent to	**bcl 12,**BI[3],target						**bltl**
bltla	**bltla** crS,target		equivalent to	**bcla 12,**BI[3],target						**bltla**
bltlr	**bltlr** crS,target		equivalent to	**bclr 12,**BI[3],target						**bltlr**
bltlrl	**bltlrl** crS,target		equivalent to	**bclrl 12,**BI[3],target						**bltlrl**
bne	**bne** crS,target		equivalent to	**bc 4,**BI[3],target						**bne**
bnea	**bnea** crS,target		equivalent to	**bca 4,**BI[3],target						**bnea**
bnectr	**bnectr** crS,target		equivalent to	**bcctr 4,**BI[3],target						**bnectr**
bnectrl	**bnectrl** crS,target		equivalent to	**bcctrl 4,**BI[3],target						**bnectrl**
bnel	**bnel** crS,target		equivalent to	**bcl 4,**BI[3],target						**bnel**
bnela	**bnela** crS,target		equivalent to	**bcla 4,**BI[3],target						**bnela**
bnelr	**bnelr** crS,target		equivalent to	**bclr 4,**BI[3],target						**bnelr**
bnelrl	**bnelrl** crS,target		equivalent to	**bclrl 4,**BI[3],target						**bnelrl**
bng	**bng** crS,target		equivalent to	**bc 4,**BI[4],target						**bng**
bnga	**bnga** crS,target		equivalent to	**bca 4,**BI[4],target						**bnga**
bngctr	**bngctr** crS,target		equivalent to	**bcctr 4,**BI[4],target						**bngctr**
bngctrl	**bngctrl** crS,target		equivalent to	**bcctrl 4,**BI[4],target						**bngctrl**
bngl	**bngl** crS,target		equivalent to	**bcl 4,**BI[4],target						**bngl**
bngla	**bngla** crS,target		equivalent to	**bcla 4,**BI[4],target						**bngla**
bnglr	**bnglr** crS,target		equivalent to	**bclr 4,**BI[4],target						**bnglr**
bnglrl	**bnglrl** crS,target		equivalent to	**bclrl 4,**BI[4],target						**bnglrl**
bnl	**bnl** crS,target		equivalent to	**bc 4,**BI[3],target						**bnl**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
bnla	**bnla crS**,target		equivalent to	**bca 4,**BI[3],target						**bnla**
bnlctr	**bnlctr crS**,target		equivalent to	**bcctr 4,**BI[3],target						**bnlctr**
bnlctrl	**bnlctrl crS**,target		equivalent to	**bcctrl 4,**BI[3],target						**bnlctrl**
bnll	**bnll crS**,target		equivalent to	**bcl 4,**BI[3],target						**bnll**
bnlla	**bnlla crS**,target		equivalent to	**bcla 4,**BI[3],target						**bnlla**
bnllr	**bnllr crS**,target		equivalent to	**bclr 4,**BI[3],target						**bnllr**
bnllrl	**bnllrl crS**,target		equivalent to	**bclrl 4,**BI[3],target						**bnllrl**
bns	**bns crS**,target		equivalent to	**bc 4,**BI[5],target						**bns**
bnsa	**bnsa crS**,target		equivalent to	**bca 4,**BI[5],target						**bnsa**
bnsctr	**bnsctr crS**,target		equivalent to	**bcctr 4,**BI[5],target						**bnsctr**
bnsctrl	**bnsctrl crS**,target		equivalent to	**bcctrl 4,**BI[5],target						**bnsctrl**
bnsl	**bnsl crS**,target		equivalent to	**bcl 4,**BI[5],target						**bnsl**
bnsla	**bnsla crS**,target		equivalent to	**bcla 4,**BI[5],target						**bnsla**
bnslr	**bnslr crS**,target		equivalent to	**bclr 4,**BI[5],target						**bnslr**
bnslrl	**bnslrl crS**,target		equivalent to	**bclrl 4,**BI[5],target						**bnslrl**
bnu	**bnu crS**,target		equivalent to	**bc 4,**BI[5],target						**bnu**
bnua	**bnua crS**,target		equivalent to	**bca 4,**BI[5],target						**bnua**
bnuctr	**bnuctr crS**,target		equivalent to	**bcctr 4,**BI[5],target						**bnuctr**
bnuctrl	**bnuctrl crS**,target		equivalent to	**bcctrl 4,**BI[5],target						**bnuctrl**
bnul	**bnul crS**,target		equivalent to	**bcl 4,**BI[5],target						**bnul**
bnula	**bnula crS**,target		equivalent to	**bcla 4,**BI[5],target						**bnula**
bnulr	**bnulr crS**,target		equivalent to	**bclr 4,**BI[5],target						**bnulr**
bnulrl	**bnulrl crS**,target		equivalent to	**bclrl 4,**BI[5],target						**bnulrl**
bso	**bso crS**,target		equivalent to	**bc 12,**BI[5],target						**bso**
bsoa	**bsoa crS**,target		equivalent to	**bca 12,**BI[5],target						**bsoa**
bsoctr	**bsoctr crS**,target		equivalent to	**bcctr 12,**BI[5],target						**bsoctr**
bsoctrl	**bsoctrl crS**,target		equivalent to	**bcctrl 12,**BI[5],target						**bsoctrl**
bsol	**bsol crS**,target		equivalent to	**bcl 12,**BI[5],target						**bsol**
bsola	**bsola crS**,target		equivalent to	**bcla 12,**BI[5],target						**bsola**
bsolr	**bsolr crS**,target		equivalent to	**bclr 12,**BI[5],target						**bsolr**
bsolrl	**bsolrl crS**,target		equivalent to	**bclrl 12,**BI[5],target						**bsolrl**
bt	**bt BI,**target		equivalent to	**bc 12,BI,**target						**bt**
bta	**bta BI,**target		equivalent to	**bca 12,BI,**target						**bta**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
btctr	**btctr BI** equivalent to **bcctr 12,BI**									**btctr**
btctrl	**btctrl BI** equivalent to **bcctrl 12,BI**									**btctrl**
btl	**btl BI,**target equivalent to **bcl 12,BI,**target									**btl**
btla	**btla BI,**target equivalent to **bcla 12,BI,**target									**btla**
btlr	**btlr BI** equivalent to **bclr 12,BI**									**btlr**
btlrl	**btlrl BI** equivalent to **bclrl 12,BI**									**btlrl**
bun	**bun crS,**target equivalent to **bc 12,**BI[5]**,**target									**bun**
buna	**buna crS,**target equivalent to **bca 12,**BI[5]**,**target									**buna**
bunctr	**bunctr crS,**target equivalent to **bcctr 12,**BI[5]**,**target									**bunctr**
bunctrl	**bunctrl crS,**target equivalent to **bcctrl 12,**BI[5]**,**target									**bunctrl**
bunl	**bunl crS,**target equivalent to **bcl 12,**BI[5]**,**target									**bunl**
bunla	**bunla crS,**target equivalent to **bcla 12,**BI[5]**,**target									**bunla**
bunlr	**bunlr crS,**target equivalent to **bclr 12,**BI[5]**,**target									**bunlr**
bunlrl	**bunlrl crS,**target equivalent to **bclrl 12,**BI[5]**,**target									**bunlrl**
clrlslwi	**clrlslwi rA,rS,***b***,***n* (*n* ≤ *b* ≤ 31) equivalent to **rlwinm rA,rS,***n***,***b* − *n*,31 − *n*									**clrlslwi**
clrlwi	**clrlwi rA,rS,***n* (n < 32) equivalent to **rlwinm rA,rS,0,***n***,31**									**clrlwi**
clrrwi	**clrrwi rA,rS,***n* (n < 32) equivalent to **rlwinm rA,rS,0,0,31** − *n*									**clrrwi**
cmp	31 (0x1F)	crfD / L	rA	rB	0 0 0	0 0 0 0	0 0 0	/	X	**cmp**
cmpi	11 (0x0B)	crfD / L	rA	SIMM					D	**cmpi**
cmpl	31 (0x1F)	/ L	rA	rB	///	0 0 0	0 1 0 0	0 0 0 /	X	**cmpl**
cmpli	10 (0x0A)	crfD / L	rA	UIMM					D	**cmpli**
cmplw	**cmplw crD,rA,rB** equivalent to **cmpl crD,0,rA,rB**									**cmplw**
cmplwi	**cmplwi crD,rA,UIMM** equivalent to **cmpli crD,0,rA,**UIMM									**cmplwi**
cmpw	**cmpw crD,rA,rB** equivalent to **cmp crD,0,rA,rB**									**cmpw**
cmpwi	**cmpwi crD,rA,SIMM** equivalent to **cmpi crD,0,rA,**SIMM									**cmpwi**
cntlzw	31 (0x1F)	rS	rA	///	0 0 0	0 0 1 1	0 1 0	0	X	**cntlzw**
cntlzw.	31 (0x1F)	rS	rA	///	0 0 0	0 0 1 1	0 1 0	1	X	**cntlzw.**
crand	19 (0x13)	crbD	crbA	crbB	0 1 0	0 0 0 0	0 0 1	/	XL	**crand**
crandc	19 (0x13)	crbD	crbA	crbB	0 0 1	0 0 0 0	0 0 1	/	XL	**crandc**
crclr	**crclr bx** equivalent to **crxor** bx,bx,bx									**crclr**
creqv	19 (0x13)	crbD	crbA	crbB	0 1 0	0 1 0 0	0 0 1	/	XL	**creqv**
crmove	**crmove bx,by** equivalent to **cror** bx,by,by									**crmove**
crnand	19 (0x13)	crbD	crbA	crbB	0 0 1	1 1 0 0	0 0 1	/	XL	**crnand**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
crnor	19 (0x13)	crbD	crbA	crbB		0 0 0	0 1 0 0	0 0 1 /	XL	**crnor**
crnot	**crnot bx,by**		equivalent to	**crnor** bx,by,by						**crnot**
cror	19 (0x13)	crbD	crbA	crbB		0 1 1	1 0 0 0	0 0 1 /	XL	**cror**
crorc	19 (0x13)	crbD	crbA	crbB		0 1 1	0 1 0 0	0 0 1 /	XL	**crorc**
crset	**crset bx**		equivalent to	**creqv** bx,bx,bx						**crset**
crxor	19 (0x13)	crbD	crbA	crbB		0 0 1	1 0 0 0	0 0 1 /	XL	**crxor**
dcba[6]	31 (0x1F)	///	rA	rB		1 0 1	1 1 1 0	1 1 0 /	X	**dcba**
dcbf	31 (0x1F)	///	rA	rB		0 0 0	1 0 1 0	1 1 0 /	X	**dcbf**
dcbi [7]	31 (0x1F)	///	rA	rB		0 1 1	1 0 1 0	1 1 0 /	X	**dcbi**
dcbst	31 (0x1F)	///	rA	rB		0 0 0	0 1 1 0	1 1 0 /	X	**dcbst**
dcbt	31 (0x1F)	CT	rA	rB		0 1 0	0 0 1 0	1 1 0 /	X	**dcbt**
dcbtst	31 (0x1F)	CT	rA	rB		0 0 1	1 1 1 0	1 1 0 /	X	**dcbtst**
dcbz	31 (0x1F)	///	rA	rB		1 1 1	1 1 1 0	1 1 0 /	X	**dcbz**
divw	31 (0x1F)	rD	rA	rB		0 1 1	1 1 0 1	0 1 1 0	X	**divw**
divw.	31 (0x1F)	rD	rA	rB		0 1 1	1 1 0 1	0 1 1 1	X	**divw.**
divwo	31 (0x1F)	rD	rA	rB		1 1 1	1 1 0 1	0 1 1 0	X	**divwo**
divwo.	31 (0x1F)	rD	rA	rB		1 1 1	1 1 0 1	0 1 1 1	X	**divwo.**
divwu	31 (0x1F)	rD	rA	rB		0 1 1	1 0 0 1	0 1 1 0	X	**divwu**
divwu.	31 (0x1F)	rD	rA	rB		0 1 1	1 0 0 1	0 1 1 1	X	**divwu.**
divwuo	31 (0x1F)	rD	rA	rB		1 1 1	1 0 0 1	0 1 1 0	X	**divwuo**
divwuo.	31 (0x1F)	rD	rA	rB		1 1 1	1 0 0 1	0 1 1 1	X	**divwuo.**
dss	**dss STRM**		equivalent to	**dss** STRM,**0**						**dss**
eciwx [6]	31 (0x1F)	rD	rA	rB		0 1 0	0 1 1 0	1 1 0 0	X	**eciwx**
ecowx [6]	31 (0x1F)	rS	rA	rB		0 1 1	0 1 1 0	1 1 0 0	X	**ecowx**
eieio	31 (0x1F)	///	///	///		1 1 0	1 0 1 0	1 1 0 0	X	**eieio**
eqv	31 (0x1F)	rD	rA	rB		0 1 0	0 0 1 1	1 0 0 0	X	**eqv**
eqv.	31 (0x1F)	rD	rA	rB		0 1 0	0 0 1 1	1 0 0 1	X	**eqv.**
extlwi	**extlwi rA,rS,*n*,*b*** (n > 0)		equivalent to	**rlwinm rA,rS,*b*,0,*n* – 1**						**extlwi**
extrwi	**extrwi rA,rS,*n*,*b* (*n* > 0)**		equivalent to	**rlwinm rA,rS,*b* + *n*, 32 – *n*,31**						**extrwi**
extsb	31 (0x1F)	rS	rA	///		1 1 1	0 1 1 1	0 1 0 0	X	**extsb**
extsb.	31 (0x1F)	rS	rA	///		1 1 1	0 1 1 1	0 1 0 1	X	**extsb.**
extsh	31 (0x1F)	rS	rA	///		1 1 1	0 0 1 1	0 1 0 0	X	**extsh**
extsh.	31 (0x1F)	rS	rA	///		1 1 1	0 0 1 1	0 1 0 1	X	**extsh.**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
fabs	63(0x3F)	frD	///	frB	0 1 0	0 0 0 1	0 0 0 0	X	**fabs**
fabs.	63(0x3F)	frD	///	frB	0 1 0	0 0 0 1	0 0 0 1	X	**fabs.**
fadd	63(0x3F)	frD	frA	frB	///	1 0	1 0 1 0	A	**fadd**
fadd.	63(0x3F)	frD	frA	frB	///	1 0	1 0 1 1	A	**fadd.**
fadds	59(0x3B)	frD	frA	frB	///	1 0	1 0 1 0	A	**fadds**
fadds.	59(0x3B)	frD	frA	frB	///	1 0	1 0 1 1	A	**fadds.**
fcmpo	63(0x3F)	crfD //	frA	frB	0 0 0	0 1 0 0	0 0 0 /	X	**fcmpo**
fcmpu	63(0x3F)	crfD //	frA	frB	0 0 0	0 0 0 0	0 0 0 /	X	**fcmpu**
fctiw	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 1 0 0	X	**fctiw**
fctiw.	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 1 0 1	X	**fctiw.**
fctiwz	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 1 1 0	X	**fctiwz**
fctiwz.	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 1 1 1	X	**fctiwz.**
fdiv	63(0x3F)	frD	frA	frB	///	1 0	0 1 0 0	A	**fdiv**
fdiv.	63(0x3F)	frD	frA	frB	///	1 0	0 1 0 1	A	**fdiv.**
fdivs	59(0x3B)	frD	frA	frB	///	1 0	0 1 0 0	A	**fdivs**
fdivs.	59(0x3B)	frD	frA	frB	///	1 0	0 1 0 1	A	**fdivs.**
fmadd	63(0x3F)	frD	frA	frB	frC	1 1	1 0 1 0	A	**fmadd**
fmadd.	63(0x3F)	frD	frA	frB	frC	1 1	1 0 1 1	A	**fmadd.**
fmadds	59(0x3B)	frD	frA	frB	frC	1 1	1 0 1 0	A	**fmadds**
fmadds.	59(0x3B)	frD	frA	frB	frC	1 1	1 0 1 1	A	**fmadds.**
fmr	63(0x3F)	frD	///	frB	0 0 0	1 0 0 1	0 0 0 0	A	**fmr**
fmr.	63(0x3F)	frD	///	frB	0 0 0	1 0 0 1	0 0 0 1	A	**fmr.**
fmsub	63(0x3F)	frD	frA	frB	frC	1 1	1 0 0 0	A	**fmsub**
fmsub.	63(0x3F)	frD	frA	frB	frC	1 1	1 0 0 1	A	**fmsub.**
fmsubs	59(0x3B)	frD	frA	frB	frC	1 1	1 0 0 0	A	**fmsubs**
fmsubs.	59(0x3B)	frD	frA	frB	frC	1 1	1 0 0 1	A	**fmsubs.**
fmul	63(0x3F)	frD	frA	///	frC	1 1	0 0 1 0	A	**fmul**
fmul.	63(0x3F)	frD	frA	///	frC	1 1	0 0 1 1	A	**fmul.**
fmuls	59(0x3B)	frD	frA	///	frC	1 1	0 0 1 0	A	**fmuls**
fmuls.	59(0x3B)	frD	frA	///	frC	1 1	0 0 1 1	A	**fmuls.**
fnabs	63(0x3F)	frD	///	frB	0 0 1	0 0 0 1	0 0 0 0	X	**fnabs**
fnabs.	63(0x3F)	frD	///	frB	0 0 1	0 0 0 1	0 0 0 1	X	**fnabs.**
fneg	63(0x3F)	frD	///	frB	0 0 0	0 1 0 1	0 0 0 0	X	**fneg**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
fneg.	63(0x3F)	frD	///	frB	0 0 0	0 1 0 1		0 0 0 1	X	**fneg.**
fnmadd	63(0x3F)	frD	frA	frB	frC		1 1	1 1 1 0	A	**fnmadd**
fnmadd.	63(0x3F)	frD	frA	frB	frC		1 1	1 1 1 1	A	**fnmadd.**
fnmadds	59(0x3B)	frD	frA	frB	frC		1 1	1 1 1 0	A	**fnmadds**
fnmadds.	59(0x3B)	frD	frA	frB	frC		1 1	1 1 1 1	A	**fnmadds.**
fnmsub	63(0x3F)	frD	frA	frB	frC		1 1	1 1 0 0	A	**fnmsub**
fnmsub.	63(0x3F)	frD	frA	frB	frC		1 1	1 1 0 1	A	**fnmsub.**
fnmsubs	59(0x3B)	frD	frA	frB	frC		1 1	1 1 0 0	A	**fnmsubs**
fnmsubs.	59(0x3B)	frD	frA	frB	frC		1 1	1 1 0 1	A	**fnmsubs.**
fres [6]	59(0x3B)	frD	///	frB	///		1 1	0 0 0 0	A	**fres**
fres. [6]	59(0x3B)	frD	///	frB	///		1 1	0 0 0 1	A	**fres.**
frsp	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1		1 0 0 0	X	**frsp**
frsp.	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1		1 0 0 1	X	**frsp.**
frsqrte [6]	63(0x3F)	frD	///	frB	///		1 1	0 1 0 0	A	**frsqrte**
frsqrte. [6]	63(0x3F)	frD	///	frB	///		1 1	0 1 0 1	A	**frsqrte.**
fsel [6]	63(0x3F)	frD	frA	frB	frC		1 0	1 1 1 0	A	**fsel**
fsel. [6]	63(0x3F)	frD	frA	frB	frC		1 0	1 1 1 1	A	**fsel.**
fsqrt [6]	63(0x3F)	frD	///	frB	///		1 0	1 1 0 0	A	**fsqrt**
fsqrt. [6]	63(0x3F)	frD	///	frB	///		1 0	1 1 0 1	A	**fsqrt.**
fsqrts [6]	59(0x3B)	frD	///	frB	///		1 0	1 1 0 0	A	**fsqrts**
fsqrts. [6]	59(0x3B)	frD	///	frB	///		1 0	1 1 0 1	A	**fsqrts.**
fsub	63(0x3F)	frD	frA	frB	///		1 0	1 0 0 0	A	**fsub**
fsub.	63(0x3F)	frD	frA	frB	///		1 0	1 0 0 1	A	**fsub.**
fsubs	59(0x3B)	frD	frA	frB	///		1 0	1 0 0 0	A	**fsubs**
fsubs.	59(0x3B)	frD	frA	frB	///		1 0	1 0 0 1	A	**fsubs.**
icbi	31 (0x1F)	///	rA	rB	1 1 1	1 0 1 0		1 1 0 /	X	**icbi**
inslwi	**inslwi rA,rS,*n*,*b* (*n* > 0)**		equivalent to	**rlwimi rA,rS,32 – *b*,*b*,(*b* + *n*) – 1**						**inslwi**
insrwi	**insrwi rA,rS,*n*,*b* (*n* > 0)**		equivalent to	**rlwimi rA,rS,32 – (*b* + *n*),*b*,(*b* + *n*) – 1**						**insrwi**
isync	19 (0x13)		///		0 0 1	0 0 1 0		1 1 0 /	XL	**isync**
la	**la rD,d(rA)**		equivalent to	**addi rD,rA,d**						**la**
lbz	34(0x22)	rD	rA	D					D	**lbz**
lbzu	35(0x23)	rD	rA	D					D	**lbzu**
lbzux	31 (0x1F)	rD	rA	rB	0 0 0	1 1 1 0		1 1 1 /	X	**lbzux**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
lbzx	31 (0x1F)	rD		rA	rB	0 0 0	1 0 1 0	1 1 1 /	X	**lbzx**
lfd	50(0x32)	frD		rA	D				D	**lfd**
lfdu	51(0x33)	frD		rA	D				D	**lfdu**
lfdux	31 (0x1F)	frD		rA	rB	1 0 0	1 1 1 0	1 1 1 /	X	**lfdux**
lfdx	31 (0x1F)	frD		rA	rB	1 0 0	1 0 1 0	1 1 1 /	X	**lfdx**
lfs	48(0x30)	frD		rA	D				D	**lfs**
lfsu	49(0x31)	frD		rA	D				D	**lfsu**
lfsux	31 (0x1F)	frD		rA	rB	1 0 0	0 1 1 0	1 1 1 /	X	**lfsux**
lfsx	31 (0x1F)	frD		rA	rB	1 0 0	0 0 1 0	1 1 1 /	X	**lfsx**
lha	42(0x2A)	rD		rA	D				D	**lha**
lhau	43(0x2B)	rD		rA	D				D	**lhau**
lhaux	31 (0x1F)	rD		rA	rB	0 1 0	1 1 1 0	1 1 1 /	X	**lhaux**
lhax	31 (0x1F)	rD		rA	rB	0 1 0	1 0 1 0	1 1 1 /	X	**lhax**
lhbrx	31 (0x1F)	rD		rA	rB	1 1 0	0 0 1 0	1 1 0 /	X	**lhbrx**
lhz	40(0x28)	rD		rA	D				D	**lhz**
lhzu	41(0x29)	rD		rA	D				D	**lhzu**
lhzux	31 (0x1F)	rD		rA	rB	0 1 0	0 1 1 0	1 1 1 /	X	**lhzux**
lhzx	31 (0x1F)	rD		rA	rB	0 1 0	0 0 1 0	1 1 1 /	X	**lhzx**
li	**li rD,value**		equivalent to		**addi rD,0,**value					**li**
lis	**lis rD,value**		equivalent to		**addis rD,0,**value					**lis**
lmw	46(0x2E)	rD		rA	D				D	**lmw**
lswi	31 (0x1F)	rD		rA	NB	1 0 0	1 0 1 0	1 0 1 /	X	**lswi**
lswx	31 (0x1F)	rD		rA	rB	1 0 0	0 0 1 0	1 0 1 /	X	**lswx**
lwarx	31 (0x1F)	rD		rA	rB	0 0 0	0 0 1 0	1 0 0 /	X	**lwarx**
lwbrx	31 (0x1F)	rD		rA	rB	1 0 0	0 0 1 0	1 1 0 /	X	**lwbrx**
lwz	32 (0x20)	rD		rA	D				D	**lwz**
lwzu	33 (0x21)	rD		rA	D				D	**lwzu**
lwzux	31 (0x1F)	rD		rA	rB	0 0 0	0 1 1 0	1 1 1 /	X	**lwzux**
lwzx	31 (0x1F)	rD		rA	rB	0 0 0	0 0 1 0	1 1 1 /	X	**lwzx**
mcrf	19 (0x13)	crfD	//	crfS	///	0 0 0	0 0 0 0	0 0 0 /	XL	**mcrf**
mcrfs	63(0x3F)	crfD	//	crfS	///	0 0 0	1 0 0 0	0 0 0 /	X	**mcrfs**
mcrxr	31 (0x1F)	crfD	///			1 0 0	0 0 0 0	0 0 0 /	X	**mcrxr**
mfcr	**mtcr rS**		equivalent to		**mtcrf 0xFF,**rS					**mfcr**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20	21 22 23	24 25 26 27	28 29 30	31	Form	Mnemonic
mfcr	31 (0x1F)	rD	///			0	0 0	0 0 1 0	0 1 1	/	X	**mfcr**
mffs	63(0x3F)	frD	///			1	0 0	1 0 0 0	1 1 1	0	X	**mffs**
mffs.	63(0x3F)	frD	///			1	0 0	1 0 0 0	1 1 1	1	X	**mffs.**
mfmsr [7]	31 (0x1F)	rD	///			0	0 0	1 0 1 0	0 1 1	/	X	**mfmsr**
mf*regname*	**mf*regname* rD**		equivalent to		**mfspr rD,SPR*n***							**mf*regname***
mfspr [8]	31 (0x1F)	rD	SPR[5–9]		SPR[0–4]	0	1 0	1 0 1 0	0 1 1	/	XFX	**mfspr**
mfsr [7]	31 (0x1F)	rD	/	SR	///	1	0 0	1 0 1 0	0 1 1	0	X	**mfsr**
mfsrin [7]	31 (0x1F)	rD	///		rB	1	0 1	0 0 1 0	0 1 1	0	X	**mfsrin**
mftb	31 (0x1F)	rD	TBR[5–9]		TBR[0–4]	0	1 0	1 1 1 0	0 1 1	0	XFX	**mftb**
mr	**mr rA,rS**		equivalent to		**or rA,rS,rS**							**mr**
mtcr	**mtcr rS**		equivalent to		**mtcrf** 0xFF**,rS**							**mtcr**
mtcrf	31 (0x1F)	rS	/	CRM	/	0	0 1	0 0 1 0	0 0 0	/	XFX	**mtcrf**
mtfsb0	63(0x3F)	crbD	///			0	0 0	1 0 0 0	1 1 0	0	X	**mtfsb0**
mtfsb0.	63(0x3F)	crbD	///			0	0 0	1 0 0 0	1 1 0	1	X	**mtfsb0.**
mtfsb1	63(0x3F)	crbD	///			0	0 0	0 1 0 0	1 1 0	0	X	**mtfsb1**
mtfsb1.	63(0x3F)	crbD	///			0	0 0	0 1 0 0	1 1 0	1	X	**mtfsb1.**
mtfsf	63(0x3F)	/	FM	/	frB	1	0 1	1 0 0 0	1 1 1	0	XFL	**mtfsf**
mtfsf.	63(0x3F)	/	FM	/	frB	1	0 1	1 0 0 0	1 1 1	1	XFL	**mtfsf.**
mtfsfi	63(0x3F)	crfD	///		IMM	/	0 0 1	0 0 0 0	1 1 0	0	X	**mtfsfi**
mtfsfi.	63(0x3F)	crfD	///		IMM	/	0 0 1	0 0 0 0	1 1 0	1	X	**mtfsfi.**
mtmsr [7]	31 (0x1F)	rS	///			0	0 1	0 0 1 0	0 1 0	/	X	**mtmsr**
mt*regname*	**mt*regname* rS**		equivalent to		**mtspr SPR*n* rS**							**mt*regname***
mtspr [8]	31 (0x1F)	rS	SPR[5–9]		SPR[0–4]	0	1 1	1 0 1 0	0 1 1	/	XFX	**mtspr**
mtsr [7]	31 (0x1F)	rS	/	SR	///	0	0 1	1 0 1 0	0 1 0	0	X	**mtsr**
mtsrin [7]	31 (0x1F)	rS	///		rB	0	0 1	1 1 1 0	0 1 0	0	X	**mtsrin**
mulhw	31 (0x1F)	rD	rA		rB	/	0 0	1 0 0 1	0 1 1	0	X	**mulhw**
mulhw.	31 (0x1F)	rD	rA		rB	/	0 0	1 0 0 1	0 1 1	1	X	**mulhw.**
mulhwu	31 (0x1F)	rD	rA		rB	/	0 0	0 0 0 1	0 1 1	0	X	**mulhwu**
mulhwu.	31 (0x1F)	rD	rA		rB	/	0 0	0 0 0 1	0 1 1	1	X	**mulhwu.**
mulli	07	rD	rA		SIMM						D	**mulli**
mullw	31 (0x1F)	rD	rA		rB	0	0 1	1 1 0 1	0 1 1	0	X	**mullw**
mullw.	31 (0x1F)	rD	rA		rB	0	0 1	1 1 0 1	0 1 1	1	X	**mullw.**
mullwo	31 (0x1F)	rD	rA		rB	1	0 1	1 1 0 1	0 1 1	0	X	**mullwo**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mullwo.	31 (0x1F)	rD		rA		rB		1	0	1	1	1	0	1	0	1	1	1	X	**mullwo.**
nand	31 (0x1F)	rS		rA		rB		0	1	1	1	0	1	1	1	0	0	0	X	**nand**
nand.	31 (0x1F)	rS		rA		rB		0	1	1	1	0	1	1	1	0	0	1	X	**nand.**
neg	31 (0x1F)	rD		rA		///		0	0	0	1	1	0	1	0	0	0	0	X	**neg**
neg.	31 (0x1F)	rD		rA		///		0	0	0	1	1	0	1	0	0	0	1	X	**neg.**
nego	31 (0x1F)	rD		rA		///		1	0	0	1	1	0	1	0	0	0	0	X	**nego**
nego.	31 (0x1F)	rD		rA		///		1	0	0	1	1	0	1	0	0	0	1	X	**nego.**
nop	**nop**		equivalent to		**ori 0,0,0**															**nop**
nor	31 (0x1F)	rS		rA		rB		0	0	0	1	1	1	1	1	0	0	0	X	**nor**
nor.	31 (0x1F)	rS		rA		rB		0	0	0	1	1	1	1	1	0	0	1	X	**nor.**
not	**not rA,rS**		equivalent to		**nor rA,rS,rS**															**not**
or	31 (0x1F)	rS		rA		rB		0	1	1	0	1	1	1	1	0	0	0	X	**or**
or.	31 (0x1F)	rS		rA		rB		0	1	1	0	1	1	1	1	0	0	1	X	**or.**
orc	31 (0x1F)	rS		rA		rB		0	1	1	0	0	1	1	1	0	0	0	X	**orc**
orc.	31 (0x1F)	rS		rA		rB		0	1	1	0	0	1	1	1	0	0	1	X	**orc.**
ori	24 (0x18)	rS		rA		UIMM													D	**ori**
oris	25 (0x19)	rS		rA		UIMM													D	**oris**
rfi [7]	19 (0x13)	///						0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
rlwimi	20 (0x14)	rS		rA		SH		MB						ME				0	M	**rlwimi**
rlwimi.	20 (0x14)	rS		rA		SH		MB						ME				1	M	**rlwimi.**
rlwinm	21 (0x15)	rS		rA		SH		MB						ME				0	M	**rlwinm**
rlwinm.	21 (0x15)	rS		rA		SH		MB						ME				1	M	**rlwinm.**
rlwnm	23 (0x17)	rS		rA		rB		MB						ME				0	M	**rlwnm**
rlwnm.	23 (0x17)	rS		rA		rB		MB						ME				1	M	**rlwnm.**
rotlw	**rotlw rA,rS,rB**		equivalent to		**rlwnm rA,rS,rB,0,31**															**rotlw**
rotlwi	**rotlwi rA,rS,*n***		equivalent to		**rlwinm rA,rS,*n*,0,31**															**rotlwi**
rotrwi	**rotrwi rA,rS,*n***		equivalent to		**rlwinm rA,rS,**32 – *n*,**0,31**															**rotrwi**
sc	17 (0x11)	///															1	/	SC	**sc**
slw	31 (0x1F)	rS		rA		rB		0	0	0	0	0	1	1	0	0	0	0	X	**slw**
slw.	31 (0x1F)	rS		rA		rB		0	0	0	0	0	1	1	0	0	0	1	X	**slw.**
slwi	**slwi rA,rS,*n*** (n < 32)		equivalent to		**rlwinm rA,rS,*n*,0,**31 – *n*															**slwi**
sraw	31 (0x1F)	rS		rA		rB		1	1	0	0	0	1	1	0	0	0	0	X	**sraw**
sraw.	31 (0x1F)	rS		rA		rB		1	1	0	0	0	1	1	0	0	0	1	X	**sraw.**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
srawi	31 (0x1F)	rS	rA	SH	1 1 0	0 1 1 1	0 0 0	0	X	**srawi**
srawi.	31 (0x1F)	rS	rA	SH	1 1 0	0 1 1 1	0 0 0	1	X	**srawi.**
srw	31 (0x1F)	rS	rA	rB	1 0 0	0 0 1 1	0 0 0	0	X	**srw**
srw.	31 (0x1F)	rS	rA	rB	1 0 0	0 0 1 1	0 0 0	1	X	**srw.**
srwi	**srwi rA,rS,***n* (n < 32)		equivalent to	**rlwinm rA,rS,32 – *n*,*n*,31**						**srwi**
stb	38(0x26)	rS	rA	D					D	**stb**
stbu	39(0x27)	rS	rA	D					D	**stbu**
stbux	31 (0x1F)	rS	rA	rB	0 0 1	1 1 1 0	1 1 1	0	X	**stbux**
stbx	31 (0x1F)	rS	rA	rB	0 0 1	1 0 1 0	1 1 1	0	X	**stbx**
stfd	54(0x36)	frS	rA	D					D	**stfd**
stfdu	55(0x37)	frS	rA	D					D	**stfdu**
stfdux	31 (0x1F)	frS	rA	rB	1 0 1	1 1 1 0	1 1 1	/	X	**stfdux**
stfdx	31 (0x1F)	frS	rA	rB	1 0 1	1 0 1 0	1 1 1	/	X	**stfdx**
stfiwx [6]	31 (0x1F)	frS	rA	rB	1 1 1	1 0 1 0	1 1 1	/	X	**stfiwx**
stfs	52(0x34)	frS	rA	D					D	**stfs**
stfsu	53(0x35)	frS	rA	D					D	**stfsu**
stfsux	31 (0x1F)	frS	rA	rB	1 0 1	0 1 1 0	1 1 1	/	X	**stfsux**
stfsx	31 (0x1F)	frS	rA	rB	1 0 1	0 0 1 0	1 1 1	/	X	**stfsx**
sth	44(0x2C)	rS	rA	D					D	**sth**
sthbrx	31 (0x1F)	rS	rA	rB	1 1 1	0 0 1 0	1 1 0	/	X	**sthbrx**
sthu	45(0x2D)	rS	rA	D					D	**sthu**
sthux	31 (0x1F)	rS	rA	rB	0 1 1	0 1 1 0	1 1 1	/	X	**sthux**
sthx	31 (0x1F)	rS	rA	rB	0 1 1	0 0 1 0	1 1 1	/	X	**sthx**
stmw	47(0x2F)	rS	rA	D					D	**stmw**
stswi	31 (0x1F)	rS	rA	NB	1 0 1	1 0 1 0	1 0 1	/	X	**stswi**
stswx	31 (0x1F)	rS	rA	rB	1 0 1	0 0 1 0	1 0 1	/	X	**stswx**
stw	36(0x24)	rS	rA	D					D	**stw**
stwbrx	31 (0x1F)	rS	rA	rB	1 0 1	0 0 1 0	1 1 0	/	X	**stwbrx**
stwcx.	31 (0x1F)	rS	rA	rB	0 0 1	0 0 1 0	1 1 0	1	X	**stwcx.**
stwu	37(0x25)	rS	rA	D					D	**stwu**
stwux	31 (0x1F)	rS	rA	rB	0 0 1	0 1 1 0	1 1 1	/	D	**stwux**
stwx	31 (0x1F)	rS	rA	rB	0 0 1	0 0 1 0	1 1 1	/	D	**stwx**
sub	**sub rD,rA,rB**		equivalent to	**subf rD,rB,rA**						**sub**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
subc	**subc rD,rA,rB**		equivalent to	**subfc rD,rB,rA**						**subc**
subf	31 (0x1F)	rD	rA	rB	0	0 0	0 1 0 1	0 0 0	X	**subf**
subf.	31 (0x1F)	rD	rA	rB	0	0 0	0 1 0 1	0 0 0	X	**subf.**
subfc	31 (0x1F)	rD	rA	rB	0	0 0	0 0 0 1	0 0 0	X	**subfc**
subfc.	31 (0x1F)	rD	rA	rB	0	0 0	0 0 0 1	0 0 0	X	**subfc.**
subfco	31 (0x1F)	rD	rA	rB	1	0 0	0 0 0 1	0 0 0	X	**subfco**
subfco.	31 (0x1F)	rD	rA	rB	1	0 0	0 0 0 1	0 0 0	X	**subfco.**
subfe	31 (0x1F)	rD	rA	rB	0	0 1	0 0 0 1	0 0 0	X	**subfe**
subfe.	31 (0x1F)	rD	rA	rB	0	0 1	0 0 0 1	0 0 0	X	**subfe.**
subfeo	31 (0x1F)	rD	rA	rB	1	0 1	0 0 0 1	0 0 0	X	**subfeo**
subfeo.	31 (0x1F)	rD	rA	rB	1	0 1	0 0 0 1	0 0 0	X	**subfeo.**
subfic	08	rD	rA	SIMM					D	**subfic**
subfme	31 (0x1F)	rD	rA	///	0	0 1	1 1 0 1	0 0 0	X	**subfme**
subfme.	31 (0x1F)	rD	rA	///	0	0 1	1 1 0 1	0 0 0	X	**subfme.**
subfmeo	31 (0x1F)	rD	rA	///	1	0 1	1 1 0 1	0 0 0	X	**subfmeo**
subfmeo.	31 (0x1F)	rD	rA	///	1	0 1	1 1 0 1	0 0 0	X	**subfmeo.**
subfo	31 (0x1F)	rD	rA	rB	1	0 0	0 1 0 1	0 0 0	X	**subfo**
subfo.	31 (0x1F)	rD	rA	rB	1	0 0	0 1 0 1	0 0 0	X	**subfo.**
subfze	31 (0x1F)	rD	rA	///	0	0 1	1 0 0 1	0 0 0	X	**subfze**
subfze.	31 (0x1F)	rD	rA	///	0	0 1	1 0 0 1	0 0 0	X	**subfze.**
subfzeo	31 (0x1F)	rD	rA	///	1	0 1	1 0 0 1	0 0 0	X	**subfzeo**
subfzeo.	31 (0x1F)	rD	rA	///	1	0 1	1 0 0 1	0 0 0	X	**subfzeo.**
subi	**subi rD,rA,value**		equivalent to	**addi rD,rA,**–value						**subi**
subic	**subic rD,rA,value**		equivalent to	**addic rD,rA,**–value						**subic**
subic.	**subic. rD,rA,value**		equivalent to	**addic. rD,rA,**–value						**subic.**
subis	**subis rD,rA,value**		equivalent to	**addis rD,rA,**–value						**subis**
sync	31 (0x1F)	///	///	///	1 0 0	1 0 1 0	1 1 0	0	X	**sync**
tlbia [6,7]	31 (0x1F)	///	///	///	0 1 0	1 1 1 0	0 1 0	0	X	**tlbia**
tlbie [6,7]	31 (0x1F)	///	///	rB	0 1 0	0 1 1 0	0 1 0	0	X	**tlbie**
tlbsync [6,7]	31 (0x1F)	///	///	///	1 0 0	0 1 1 0	1 1 0	/	X	**tlbsync**
tw	31 (0x1F)	TO	rA	rB	0 0 0	0 0 0 0	1 0 0	/	X	**tw**
tweq	**tweq rA,SIMM**		equivalent to	**tw 4,rA,**SIMM						**tweq**
tweqi	**tweqi rA,SIMM**		equivalent to	**twi 4,rA,**SIMM						**tweqi**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
twge	**twge** r**A**,**SIMM**	equivalent to	**tw 12,**r**A**,SIMM							**twge**
twgei	**twgei** r**A**,**SIMM**	equivalent to	**twi 12,**r**A**,SIMM							**twgei**
twgt	**twgt** r**A**,**SIMM**	equivalent to	**tw 8,**r**A**,SIMM							**twgt**
twgti	**twgti** r**A**,**SIMM**	equivalent to	**twi 8,**r**A**,SIMM							**twgti**
twi	03	TO	rA	SIMM					D	**twi**
twle	**twle** r**A**,**SIMM**	equivalent to	**tw 20,**r**A**,SIMM							**twle**
twlei	**twlei** r**A**,**SIMM**	equivalent to	**twi 20,**r**A**,SIMM							**twlei**
twlge	**twlge** r**A**,**SIMM**	equivalent to	**tw 12,**r**A**,SIMM							**twlge**
twlgei	**twlgei** r**A**,**SIMM**	equivalent to	**twi 12,**r**A**,SIMM							**twlgei**
twlgt	**twlgt** r**A**,**SIMM**	equivalent to	**tw 1,**r**A**,SIMM							**twlgt**
twlgti	**twlgti** r**A**,**SIMM**	equivalent to	**twi 1,**r**A**,SIMM							**twlgti**
twlle	**twlle** r**A**,**SIMM**	equivalent to	**tw 6,**r**A**,SIMM							**twlle**
twllei	**twllei** r**A**,**SIMM**	equivalent to	**twi 6,**r**A**,SIMM							**twllei**
twllt	**twllt** r**A**,**SIMM**	equivalent to	**tw 2,**r**A**,SIMM							**twllt**
twllti	**twllti** r**A**,**SIMM**	equivalent to	**twi 2,**r**A**,SIMM							**twllti**
twlng	**twlng** r**A**,**SIMM**	equivalent to	**tw 6,**r**A**,SIMM							**twlng**
twlngi	**twlngi** r**A**,**SIMM**	equivalent to	**twi 6,**r**A**,SIMM							**twlngi**
twlnl	**twlnl** r**A**,**SIMM**	equivalent to	**tw 5,**r**A**,SIMM							**twlnl**
twlnli	**twlnli** r**A**,**SIMM**	equivalent to	**twi 5,**r**A**,SIMM							**twlnli**
twlt	**twlt** r**A**,**SIMM**	equivalent to	**tw 16,**r**A**,SIMM							**twlt**
twlti	**twlti** r**A**,**SIMM**	equivalent to	**twi 16,**r**A**,SIMM							**twlti**
twne	**twne** r**A**,**SIMM**	equivalent to	**tw 24,**r**A**,SIMM							**twne**
twnei	**twnei** r**A**,**SIMM**	equivalent to	**twi 24,**r**A**,SIMM							**twnei**
twng	**twng** r**A**,**SIMM**	equivalent to	**tw 20,**r**A**,SIMM							**twng**
twngi	**twngi** r**A**,**SIMM**	equivalent to	**twi 20,**r**A**,SIMM							**twngi**
twnl	**twnl** r**A**,**SIMM**	equivalent to	**tw 12,**r**A**,SIMM							**twnl**
twnli	**twnli** r**A**,**SIMM**	equivalent to	**twi 12,**r**A**,SIMM							**twnli**

Table A-1. Instructions Sorted by Mnemonic (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
xor	31 (0x1F)	rS	rA	rB	0 1 0	0 1 1 1	1 0 0	0	X	**xor**
xor.	31 (0x1F)	rS	rA	rB	0 1 0	0 1 1 1	1 0 0	1	X	**xor.**
xori	26 (0x1A)	rS	rA	UIMM					D	**xori**
xoris	27 (0x1B)	rS	rA	UIMM					D	**xoris**

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.
2 The value in the BI operand selects CR*n*[2], the EQ bit.
3 The value in the BI operand selects CR*n*[0], the LT bit.
4 The value in the BI operand selects CR*n*[1], the GT bit.
5 The value in the BI operand selects CR*n*[3], the SO bit.
6 Optional to the PowerPC classic architecture.
7 Supervisor-level instruction
8 Access level is detemined by whether the SPR is defined as a user- or supervisor-level SPR.

A.2 Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal)

Table A-2 lists instructions by their primary (0–5) opcodes in decimal and hexadecimal format.

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10	11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
twi	03	TO			rA	SIMM				D	**twi**
mulli	07	rD			rA	SIMM				D	**mulli**
subfic	08	rD			rA	SIMM				D	**subfic**
cmpli	10 (0x0A)	crfD	/	L	rA	UIMM				D	**cmpli**
cmpi	11 (0x0B)	crfD	/	L	rA	SIMM				D	**cmpi**
addic	12 (0x0C)	rD			rA	SIMM				D	**addic**
addic.	13 (0x0D)	rD			rA	SIMM				D	**addic.**
addi	14 (0x0E)	rD			rA	SIMM				D	**addi**
addis	15 (0x0F)	rD			rA	SIMM				D	**addis**
bc	16 (0x10)	BO			BI	BD			0	0	B	**bc**
bca	16 (0x10)	BO			BI	BD			1	0	B	**bca**
bcl	16 (0x10)	BO			BI	BD			0	1	B	**bcl**
bcla	16 (0x10)	BO			BI	BD			1	1	B	**bcla**
sc	17 (0x11)	///							1	/	SC	**sc**
b	18 (0x12)	LI							0	0	I	**b**
ba	18 (0x12)	LI							1	0	I	**ba**
bl	18 (0x12)	LI							0	1	I	**bl**

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
bla	18 (0x12)	LI						1 1	I	**bla**
mcrf	19 (0x13)	crfD //	crfS	///		0 0 0	0 0 0 0	0 0 0 /	XL	**mcrf**
bclr	19 (0x13)	BO	BI	///		0 0 0	0 0 1 0	0 0 0 0	XL	**bclr**
bclrl	19 (0x13)	BO	BI	///		0 0 0	0 0 1 0	0 0 0 1	XL	**bclrl**
crnor	19 (0x13)	crbD	crbA	crbB		0 0 0	0 1 0 0	0 0 1 /	XL	**crnor**
rfi [1]	19 (0x13)	///				0 0 0	0 1 1 0	0 1 0 /	XL	**rfi**
crandc	19 (0x13)	crbD	crbA	crbB		0 0 1	0 0 0 0	0 0 1 /	XL	**crandc**
isync	19 (0x13)	///				0 0 1	0 0 1 0	1 1 0 /	XL	**isync**
crxor	19 (0x13)	crbD	crbA	crbB		0 0 1	1 0 0 0	0 0 1 /	XL	**crxor**
crand	19 (0x13)	crbD	crbA	crbB		0 1 0	0 0 0 0	0 0 1 /	XL	**crand**
crnand	19 (0x13)	crbD	crbA	crbB		0 0 1	1 1 0 0	0 0 1 /	XL	**crnand**
creqv	19 (0x13)	crbD	crbA	crbB		0 1 0	0 1 0 0	0 0 1 /	XL	**creqv**
crorc	19 (0x13)	crbD	crbA	crbB		0 1 1	0 1 0 0	0 0 1 /	XL	**crorc**
cror	19 (0x13)	crbD	crbA	crbB		0 1 1	1 0 0 0	0 0 1 /	XL	**cror**
bcctr	19 (0x13)	BO	BI	///		1 0 0	0 0 1 0	0 0 0 0	XL	**bcctr**
bcctrl	19 (0x13)	BO	BI	///		1 0 0	0 0 1 0	0 0 0 1	XL	**bcctrl**
rlwimi	20 (0x14)	rS	rA	SH		MB	ME	0	M	**rlwimi**
rlwimi.	20 (0x14)	rS	rA	SH		MB	ME	1	M	**rlwimi.**
rlwinm	21 (0x15)	rS	rA	SH		MB	ME	0	M	**rlwinm**
rlwinm.	21 (0x15)	rS	rA	SH		MB	ME	1	M	**rlwinm.**
rlwnm	23 (0x17)	rS	rA	rB		MB	ME	0	M	**rlwnm**
rlwnm.	23 (0x17)	rS	rA	rB		MB	ME	1	M	**rlwnm.**
ori	24 (0x18)	rS	rA	UIMM					D	**ori**
oris	25 (0x19)	rS	rA	UIMM					D	**oris**
xori	26 (0x1A)	rS	rA	UIMM					D	**xori**
xoris	27 (0x1B)	rS	rA	UIMM					D	**xoris**
andi.	28 (0x1C)	rS	rA	UIMM					D	**andi.**
andis.	29 (0x1D)	rS	rA	UIMM					D	**andis.**
cmp	31 (0x1F)	crfD / L	rA	rB		0 0 0	0 0 0 0	0 0 0 /	X	**cmp**
tw	31 (0x1F)	TO	rA	rB		0 0 0	0 0 0 0	1 0 0 /	X	**tw**
subfc	31 (0x1F)	rD	rA	rB		0 0 0	0 0 0 1	0 0 0 0	X	**subfc**
subfc.	31 (0x1F)	rD	rA	rB		0 0 0	0 0 0 1	0 0 0 1	X	**subfc.**
addc	31 (0x1F)	rD	rA	rB		0 0 0	0 0 0 1	0 1 0 0	X	**addc**

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20	21 22 23	24 25 26 27	28 29 30	31	Form	Mnemonic
addc.	31 (0x1F)	rD		rA	rB	0	0 0	0 0 0 1	0 1 0	1	X	addc.
mulhwu	31 (0x1F)	rD		rA	rB	/	0 0	0 0 0 1	0 1 1	0	X	mulhwu
mulhwu.	31 (0x1F)	rD		rA	rB	/	0 0	0 0 0 1	0 1 1	1	X	mulhwu.
mfcr	31 (0x1F)	rD		///		0	0 0	0 0 1 0	0 1 1	/	X	mfcr
lwarx	31 (0x1F)	rD		rA	rB	0	0 0	0 0 1 0	1 0 0	/	X	lwarx
lwzx	31 (0x1F)	rD		rA	rB	0	0 0	0 0 1 0	1 1 1	/	X	lwzx
slw	31 (0x1F)	rS		rA	rB	0	0 0	0 0 1 1	0 0 0	0	X	slw
slw.	31 (0x1F)	rS		rA	rB	0	0 0	0 0 1 1	0 0 0	1	X	slw.
cntlzw	31 (0x1F)	rS		rA	///	0	0 0	0 0 1 1	0 1 0	0	X	cntlzw
cntlzw.	31 (0x1F)	rS		rA	///	0	0 0	0 0 1 1	0 1 0	1	X	cntlzw.
and	31 (0x1F)	rS		rA	rB	0	0 0	0 0 1 1	1 0 0	0	X	and
and.	31 (0x1F)	rS		rA	rB	0	0 0	0 0 1 1	1 0 0	1	X	and.
cmpl	31 (0x1F)	/	L	rA	rB	///	0	0 0 0	0 1 0 0	0 0 0	/	X	cmpl
subf	31 (0x1F)	rD		rA	rB	0	0 0	0 1 0 1	0 0 0	0	X	subf
subf.	31 (0x1F)	rD		rA	rB	0	0 0	0 1 0 1	0 0 0	1	X	subf.
dcbst	31 (0x1F)	///		rA	rB	0	0 0	0 1 1 0	1 1 0	/	X	dcbst
lwzux	31 (0x1F)	rD		rA	rB	0	0 0	0 1 1 0	1 1 1	/	X	lwzux
andc	31 (0x1F)	rS		rA	rB	0	0 0	0 1 1 1	1 0 0	0	X	andc
andc.	31 (0x1F)	rS		rA	rB	0	0 0	0 1 1 1	1 0 0	1	X	andc.
mulhw	31 (0x1F)	rD		rA	rB	/	0 0	1 0 0 1	0 1 1	0	X	mulhw
mulhw.	31 (0x1F)	rD		rA	rB	/	0 0	1 0 0 1	0 1 1	1	X	mulhw.
mfmsr [1]	31 (0x1F)	rD		///		0	0 0	1 0 1 0	0 1 1	/	X	mfmsr
dcbf	31 (0x1F)	///		rA	rB	0	0 0	1 0 1 0	1 1 0	/	X	dcbf
lbzx	31 (0x1F)	rD		rA	rB	0	0 0	1 0 1 0	1 1 1	/	X	lbzx
neg	31 (0x1F)	rD		rA	///	0	0 0	1 1 0 1	0 0 0	0	X	neg
neg.	31 (0x1F)	rD		rA	///	0	0 0	1 1 0 1	0 0 0	1	X	neg.
lbzux	31 (0x1F)	rD		rA	rB	0	0 0	1 1 1 0	1 1 1	/	X	lbzux
nor	31 (0x1F)	rS		rA	rB	0	0 0	1 1 1 1	1 0 0	0	X	nor
nor.	31 (0x1F)	rS		rA	rB	0	0 0	1 1 1 1	1 0 0	1	X	nor.
subfe	31 (0x1F)	rD		rA	rB	0	0 1	0 0 0 1	0 0 0	0	X	subfe
subfe.	31 (0x1F)	rD		rA	rB	0	0 1	0 0 0 1	0 0 0	1	X	subfe.
adde	31 (0x1F)	rD		rA	rB	0	0 1	0 0 0 1	0 1 0	0	X	adde
adde.	31 (0x1F)	rD		rA	rB	0	0 1	0 0 0 1	0 1 0	1	X	adde.

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10	11	12 13 14 15	16 17 18 19	20	21 22 23	24 25 26 27	28 29 30	31	Form	Mnemonic
mtcrf	31 (0x1F)	rS		/	CRM		/	0 0 1	0 0 1 0	0 0 0	/	XFX	**mtcrf**
mtmsr [1]	31 (0x1F)	rS		///				0 0 1	0 0 1 0	0 1 0	/	X	**mtmsr**
stwcx.	31 (0x1F)	rS		rA		rB		0 0 1	0 0 1 0	1 1 0	1	X	**stwcx.**
stwx	31 (0x1F)	rS		rA		rB		0 0 1	0 0 1 0	1 1 1	/	D	**stwx**
stwux	31 (0x1F)	rS		rA		rB		0 0 1	0 1 1 0	1 1 1	/	D	**stwux**
subfze	31 (0x1F)	rD		rA		///		0 0 1	1 0 0 1	0 0 0	0	X	**subfze**
subfze.	31 (0x1F)	rD		rA		///		0 0 1	1 0 0 1	0 0 0	1	X	**subfze.**
addze	31 (0x1F)	rD		rA		///		0 0 1	1 0 0 1	0 1 0	0	X	**addze**
addze.	31 (0x1F)	rD		rA		///		0 0 1	1 0 0 1	0 1 0	1	X	**addze.**
mtsr [1]	31 (0x1F)	rS		/	SR	///		0 0 1	1 0 1 0	0 1 0	0	X	**mtsr**
stbx	31 (0x1F)	rS		rA		rB		0 0 1	1 0 1 0	1 1 1	0	X	**stbx**
subfme	31 (0x1F)	rD		rA		///		0 0 1	1 1 0 1	0 0 0	0	X	**subfme**
subfme.	31 (0x1F)	rD		rA		///		0 0 1	1 1 0 1	0 0 0	1	X	**subfme.**
addme	31 (0x1F)	rD		rA		///		0 0 1	1 1 0 1	0 1 0	0	X	**addme**
addme.	31 (0x1F)	rD		rA		///		0 0 1	1 1 0 1	0 1 0	1	X	**addme.**
mullw	31 (0x1F)	rD		rA		rB		0 0 1	1 1 0 1	0 1 1	0	X	**mullw**
mullw.	31 (0x1F)	rD		rA		rB		0 0 1	1 1 0 1	0 1 1	1	X	**mullw.**
mtsrin [1]	31 (0x1F)	rS		///		rB		0 0 1	1 1 1 0	0 1 0	0	X	**mtsrin**
dcbtst	31 (0x1F)	CT		rA		rB		0 0 1	1 1 1 0	1 1 0	/	X	**dcbtst**
stbux	31 (0x1F)	rS		rA		rB		0 0 1	1 1 1 0	1 1 1	0	X	**stbux**
add	31 (0x1F)	rD		rA		rB		0 1 0	0 0 0 1	0 1 0	0	X	**add**
add.	31 (0x1F)	rD		rA		rB		0 1 0	0 0 0 1	0 1 0	1	X	**add.**
dcbt	31 (0x1F)	CT		rA		rB		0 1 0	0 0 1 0	1 1 0	/	X	**dcbt**
lhzx	31 (0x1F)	rD		rA		rB		0 1 0	0 0 1 0	1 1 1	/	X	**lhzx**
eqv	31 (0x1F)	rD		rA		rB		0 1 0	0 0 1 1	1 0 0	0	X	**eqv**
eqv.	31 (0x1F)	rD		rA		rB		0 1 0	0 0 1 1	1 0 0	1	X	**eqv.**
tlbie [1, 2]	31 (0x1F)	///		///		rB		0 1 0	0 1 1 0	0 1 0	0	X	**tlbie**
eciwx [2]	31 (0x1F)	rD		rA		rB		0 1 0	0 1 1 0	1 1 0	0	X	**eciwx**
lhzux	31 (0x1F)	rD		rA		rB		0 1 0	0 1 1 0	1 1 1	/	X	**lhzux**
xor	31 (0x1F)	rS		rA		rB		0 1 0	0 1 1 1	1 0 0	0	X	**xor**
xor.	31 (0x1F)	rS		rA		rB		0 1 0	0 1 1 1	1 0 0	1	X	**xor.**
mfspr [3]	31 (0x1F)	rD		SPR[5–9]		SPR[0–4]		0 1 0	1 0 1 0	0 1 1	/	XFX	**mfspr**
lhax	31 (0x1F)	rD		rA		rB		0 1 0	1 0 1 0	1 1 1	/	X	**lhax**

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10	11	12 13 14 15	16 17 18 19	20	21 22 23	24 25 26 27	28 29 30	31	Form	Mnemonic
tlbia [1, 2]	31 (0x1F)	///		///		///	0	1 0	1 1 1 0	0 1 0	0	X	**tlbia**
mftb	31 (0x1F)	rD		TBR[5–9]		TBR[0–4]	0	1 0	1 1 1 0	0 1 1	0	XFX	**mftb**
lhaux	31 (0x1F)	rD		rA		rB	0	1 0	1 1 1 0	1 1 1	/	X	**lhaux**
sthx	31 (0x1F)	rS		rA		rB	0	1 1	0 0 1 0	1 1 1	/	X	**sthx**
orc	31 (0x1F)	rS		rA		rB	0	1 1	0 0 1 1	1 0 0	0	X	**orc**
orc.	31 (0x1F)	rS		rA		rB	0	1 1	0 0 1 1	1 0 0	1	X	**orc.**
ecowx [2]	31 (0x1F)	rS		rA		rB	0	1 1	0 1 1 0	1 1 0	0	X	**ecowx**
sthux	31 (0x1F)	rS		rA		rB	0	1 1	0 1 1 0	1 1 1	/	X	**sthux**
or	31 (0x1F)	rS		rA		rB	0	1 1	0 1 1 1	1 0 0	0	X	**or**
or.	31 (0x1F)	rS		rA		rB	0	1 1	0 1 1 1	1 0 0	1	X	**or.**
divwu	31 (0x1F)	rD		rA		rB	0	1 1	1 0 0 1	0 1 1	0	X	**divwu**
divwu.	31 (0x1F)	rD		rA		rB	0	1 1	1 0 0 1	0 1 1	1	X	**divwu.**
mtspr [3]	31 (0x1F)	rS		SPR[5–9]		SPR[0–4]	0	1 1	1 0 1 0	0 1 1	/	XFX	**mtspr**
dcbi [1]	31 (0x1F)	///		rA		rB	0	1 1	1 0 1 0	1 1 0	/	X	**dcbi**
nand	31 (0x1F)	rS		rA		rB	0	1 1	1 0 1 1	1 0 0	0	X	**nand**
nand.	31 (0x1F)	rS		rA		rB	0	1 1	1 0 1 1	1 0 0	1	X	**nand.**
divw	31 (0x1F)	rD		rA		rB	0	1 1	1 1 0 1	0 1 1	0	X	**divw**
divw.	31 (0x1F)	rD		rA		rB	0	1 1	1 1 0 1	0 1 1	1	X	**divw.**
mcrxr	31 (0x1F)	crfD		///			1	0 0	0 0 0 0	0 0 0	/	X	**mcrxr**
subfco	31 (0x1F)	rD		rA		rB	1	0 0	0 0 0 1	0 0 0	0	X	**subfco**
subfco.	31 (0x1F)	rD		rA		rB	1	0 0	0 0 0 1	0 0 0	1	X	**subfco.**
addco	31 (0x1F)	rD		rA		rB	1	0 0	0 0 0 1	0 1 0	0	X	**addco**
addco.	31 (0x1F)	rD		rA		rB	1	0 0	0 0 0 1	0 1 0	1	X	**addco.**
lswx	31 (0x1F)	rD		rA		rB	1	0 0	0 0 1 0	1 0 1	/	X	**lswx**
lwbrx	31 (0x1F)	rD		rA		rB	1	0 0	0 0 1 0	1 1 0	/	X	**lwbrx**
lfsx	31 (0x1F)	frD		rA		rB	1	0 0	0 0 1 0	1 1 1	/	X	**lfsx**
srw	31 (0x1F)	rS		rA		rB	1	0 0	0 0 1 1	0 0 0	0	X	**srw**
srw.	31 (0x1F)	rS		rA		rB	1	0 0	0 0 1 1	0 0 0	1	X	**srw.**
subfo	31 (0x1F)	rD		rA		rB	1	0 0	0 1 0 1	0 0 0	0	X	**subfo**
subfo.	31 (0x1F)	rD		rA		rB	1	0 0	0 1 0 1	0 0 0	1	X	**subfo.**
tlbsync [1,2]	31 (0x1F)	///		///		///	1	0 0	0 1 1 0	1 1 0	/	X	**tlbsync**
lfsux	31 (0x1F)	frD		rA		rB	1	0 0	0 1 1 0	1 1 1	/	X	**lfsux**
mfsr [1]	31 (0x1F)	rD	/	SR		///	1	0 0	1 0 1 0	0 1 1	0	X	**mfsr**

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
lswi	31 (0x1F)	rD	rA	NB	1 0 0	1 0 1 0	1 0 1 /	X	**lswi**
sync	31 (0x1F)	///	///	///	1 0 0	1 0 1 0	1 1 0 0	X	**sync**
lfdx	31 (0x1F)	frD	rA	rB	1 0 0	1 0 1 0	1 1 1 /	X	**lfdx**
nego	31 (0x1F)	rD	rA	///	1 0 0	1 1 0 1	0 0 0 0	X	**nego**
nego.	31 (0x1F)	rD	rA	///	1 0 0	1 1 0 1	0 0 0 1	X	**nego.**
lfdux	31 (0x1F)	frD	rA	rB	1 0 0	1 1 1 0	1 1 1 /	X	**lfdux**
subfeo	31 (0x1F)	rD	rA	rB	1 0 1	0 0 0 1	0 0 0 0	X	**subfeo**
subfeo.	31 (0x1F)	rD	rA	rB	1 0 1	0 0 0 1	0 0 0 1	X	**subfeo.**
addeo	31 (0x1F)	rD	rA	rB	1 0 1	0 0 0 1	0 1 0 0	X	**addeo**
addeo.	31 (0x1F)	rD	rA	rB	1 0 1	0 0 0 1	0 1 0 1	X	**addeo.**
mfsrin [1]	31 (0x1F)	rS	///	rB	0 0 1	1 1 1 0	0 1 0 0	X	**mtsrin**
stswx	31 (0x1F)	rS	rA	rB	1 0 1	0 0 1 0	1 0 1 /	X	**stswx**
stwbrx	31 (0x1F)	rS	rA	rB	1 0 1	0 0 1 0	1 1 0 /	X	**stwbrx**
stfsx	31 (0x1F)	frS	rA	rB	1 0 1	0 0 1 0	1 1 1 /	X	**stfsx**
stfsux	31 (0x1F)	frS	rA	rB	1 0 1	0 1 1 0	1 1 1 /	X	**stfsux**
subfzeo	31 (0x1F)	rD	rA	///	1 0 1	1 0 0 1	0 0 0 0	X	**subfzeo**
subfzeo.	31 (0x1F)	rD	rA	///	1 0 1	1 0 0 1	0 0 0 1	X	**subfzeo.**
addzeo	31 (0x1F)	rD	rA	///	1 0 1	1 0 0 1	0 1 0 0	X	**addzeo**
addzeo.	31 (0x1F)	rD	rA	///	1 0 1	1 0 0 1	0 1 0 1	X	**addzeo.**
stswi	31 (0x1F)	rS	rA	NB	1 0 1	1 0 1 0	1 0 1 /	X	**stswi**
stfdx	31 (0x1F)	frS	rA	rB	1 0 1	1 0 1 0	1 1 1 /	X	**stfdx**
subfmeo	31 (0x1F)	rD	rA	///	1 0 1	1 1 0 1	0 0 0 0	X	**subfmeo**
subfmeo.	31 (0x1F)	rD	rA	///	1 0 1	1 1 0 1	0 0 0 1	X	**subfmeo.**
addmeo	31 (0x1F)	rD	rA	///	1 0 1	1 1 0 1	0 1 0 0	X	**addmeo**
addmeo.	31 (0x1F)	rD	rA	///	1 0 1	1 1 0 1	0 1 0 1	X	**addmeo.**
mullwo	31 (0x1F)	rD	rA	rB	1 0 1	1 1 0 1	0 1 1 0	X	**mullwo**
mullwo.	31 (0x1F)	rD	rA	rB	1 0 1	1 1 0 1	0 1 1 1	X	**mullwo.**
dcba [2]	31 (0x1F)	///	rA	rB	1 0 1	1 1 1 0	1 1 0 /	X	**dcba**
stfdux	31 (0x1F)	frS	rA	rB	1 0 1	1 1 1 0	1 1 1 /	X	**stfdux**
addo	31 (0x1F)	rD	rA	rB	1 1 0	0 0 0 1	0 1 0 0	X	**addo**
addo.	31 (0x1F)	rD	rA	rB	1 1 0	0 0 0 1	0 1 0 1	X	**addo.**
lhbrx	31 (0x1F)	rD	rA	rB	1 1 0	0 0 1 0	1 1 0 /	X	**lhbrx**
sraw	31 (0x1F)	rS	rA	rB	1 1 0	0 0 1 1	0 0 0 0	X	**sraw**

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
sraw.	31 (0x1F)	rS	rA	rB	1 1 0	0 0 1 1	0 0 0 1		X	sraw.
srawi	31 (0x1F)	rS	rA	SH	1 1 0	0 1 1 1	0 0 0 0		X	srawi
srawi.	31 (0x1F)	rS	rA	SH	1 1 0	0 1 1 1	0 0 0 1		X	srawi.
eieio	31 (0x1F)	///	///	///	1 1 0	1 0 1 0	1 1 0 0		X	eieio
sthbrx	31 (0x1F)	rS	rA	rB	1 1 1	0 0 1 0	1 1 0 /		X	sthbrx
extsh	31 (0x1F)	rS	rA	///	1 1 1	0 0 1 1	0 1 0 0		X	extsh
extsh.	31 (0x1F)	rS	rA	///	1 1 1	0 0 1 1	0 1 0 1		X	extsh.
extsb	31 (0x1F)	rS	rA	///	1 1 1	0 1 1 1	0 1 0 0		X	extsb
extsb.	31 (0x1F)	rS	rA	///	1 1 1	0 1 1 1	0 1 0 1		X	extsb.
divwuo	31 (0x1F)	rD	rA	rB	1 1 1	1 0 0 1	0 1 1 0		X	divwuo
divwuo.	31 (0x1F)	rD	rA	rB	1 1 1	1 0 0 1	0 1 1 1		X	divwuo.
icbi	31 (0x1F)	///	rA	rB	1 1 1	1 0 1 0	1 1 0 /		X	icbi
stfiwx [2]	31 (0x1F)	frS	rA	rB	1 1 1	1 0 1 0	1 1 1 /		X	stfiwx
divwo	31 (0x1F)	rD	rA	rB	1 1 1	1 1 0 1	0 1 1 0		X	divwo
divwo.	31 (0x1F)	rD	rA	rB	1 1 1	1 1 0 1	0 1 1 1		X	divwo.
dcbz	31 (0x1F)	///	rA	rB	1 1 1	1 1 1 0	1 1 0 /		X	dcbz
lwz	32 (0x20)	rD	rA	D					D	lwz
lwzu	33 (0x21)	rD	rA	D					D	lwzu
lbz	34(0x22)	rD	rA	D					D	lbz
lbzu	35(0x23)	rD	rA	D					D	lbzu
stw	36(0x24)	rS	rA	D					D	stw
stwu	37(0x25)	rS	rA	D					D	stwu
stb	38(0x26)	rS	rA	D					D	stb
stbu	39(0x27)	rS	rA	D					D	stbu
lhz	40(0x28)	rD	rA	D					D	lhz
lhzu	41(0x29)	rD	rA	D					D	lhzu
lha	42(0x2A)	rD	rA	D					D	lha
lhau	43(0x2B)	rD	rA	D					D	lhau
sth	44(0x2C)	rS	rA	D					D	sth
sthu	45(0x2D)	rS	rA	D					D	sthu
lmw	46(0x2E)	rD	rA	D					D	lmw
stmw	47(0x2F)	rS	rA	D					D	stmw
lfs	48(0x30)	frD	rA	D					D	lfs

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
lfsu	49(0x31)	frD	rA	D					D	**lfsu**
lfd	50(0x32)	frD	rA	D					D	**lfd**
lfdu	51(0x33)	frD	rA	D					D	**lfdu**
stfs	52(0x34)	frS	rA	D					D	**stfs**
stfsu	53(0x35)	frS	rA	D					D	**stfsu**
stfd	54(0x36)	frS	rA	D					D	**stfd**
stfdu	55(0x37)	frS	rA	D					D	**stfdu**
fdivs	59(0x3B)	frD	frA	frB	///	1 0	0 1 0	0	A	**fdivs**
fdivs.	59(0x3B)	frD	frA	frB	///	1 0	0 1 0	1	A	**fdivs.**
fsubs	59(0x3B)	frD	frA	frB	///	1 0	1 0 0	0	A	**fsubs**
fsubs.	59(0x3B)	frD	frA	frB	///	1 0	1 0 0	1	A	**fsubs.**
fadds	59(0x3B)	frD	frA	frB	///	1 0	1 0 1	0	A	**fadds**
fadds.	59(0x3B)	frD	frA	frB	///	1 0	1 0 1	1	A	**fadds.**
fsqrts [2]	59(0x3B)	frD	///	frB	///	1 0	1 1 0	0	A	**fsqrts**
fsqrts. [2]	59(0x3B)	frD	///	frB	///	1 0	1 1 0	1	A	**fsqrts.**
fres [2]	59(0x3B)	frD	///	frB	///	1 1	0 0 0	0	A	**fres**
fres. [2]	59(0x3B)	frD	///	frB	///	1 1	0 0 0	1	A	**fres.**
fmuls	59(0x3B)	frD	frA	///	frC	1 1	0 0 1	0	A	**fmuls**
fmuls.	59(0x3B)	frD	frA	///	frC	1 1	0 0 1	1	A	**fmuls.**
fmsubs	59(0x3B)	frD	frA	frB	frC	1 1	1 0 0	0	A	**fmsubs**
fmsubs.	59(0x3B)	frD	frA	frB	frC	1 1	1 0 0	1	A	**fmsubs.**
fmadds	59(0x3B)	frD	frA	frB	frC	1 1	1 0 1	0	A	**fmadds**
fmadds.	59(0x3B)	frD	frA	frB	frC	1 1	1 0 1	1	A	**fmadds.**
fnmsubs	59(0x3B)	frD	frA	frB	frC	1 1	1 1 0	0	A	**fnmsubs**
fnmsubs.	59(0x3B)	frD	frA	frB	frC	1 1	1 1 0	1	A	**fnmsubs.**
fnmadds	59(0x3B)	frD	frA	frB	frC	1 1	1 1 1	0	A	**fnmadds**
fnmadds.	59(0x3B)	frD	frA	frB	frC	1 1	1 1 1	1	A	**fnmadds.**
fcmpu	63(0x3F)	crfD //	frA	frB	0 0 0	0 0 0 0	0 0 0	/	X	**fcmpu**
frsp	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 0 0	0	X	**frsp**
frsp.	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 0 0	1	X	**frsp.**
fctiw	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 1 0	0	X	**fctiw**
fctiw.	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 1 0	1	X	**fctiw.**
fctiwz	63(0x3F)	frD	///	frB	0 0 0	0 0 0 1	1 1 1	0	X	**fctiwz**

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
fctiwz.	63(0x3F)	frD	///		frB	0 0 0	0 0 0 1	1 1 1 1	X	**fctiwz.**
fdiv	63(0x3F)	frD	frA		frB	///	1 0	0 1 0 0	A	**fdiv**
fdiv.	63(0x3F)	frD	frA		frB	///	1 0	0 1 0 1	A	**fdiv.**
fsub	63(0x3F)	frD	frA		frB	///	1 0	1 0 0 0	A	**fsub**
fsub.	63(0x3F)	frD	frA		frB	///	1 0	1 0 0 1	A	**fsub.**
fadd	63(0x3F)	frD	frA		frB	///	1 0	1 0 1 0	A	**fadd**
fadd.	63(0x3F)	frD	frA		frB	///	1 0	1 0 1 1	A	**fadd.**
fsqrt [2]	63(0x3F)	frD	///		frB	///	1 0	1 1 0 0	A	**fsqrt**
fsqrt. [2]	63(0x3F)	frD	///		frB	///	1 0	1 1 0 1	A	**fsqrt.**
fsel [2]	63(0x3F)	frD	frA		frB	frC	1 0	1 1 1 0	A	**fsel**
fsel. [2]	63(0x3F)	frD	frA		frB	frC	1 0	1 1 1 1	A	**fsel.**
fmul	63(0x3F)	frD	frA		///	frC	1 1	0 0 1 0	A	**fmul**
fmul.	63(0x3F)	frD	frA		///	frC	1 1	0 0 1 1	A	**fmul.**
frsqrte [2]	63(0x3F)	frD	///		frB	///	1 1	0 1 0 0	A	**frsqrte**
frsqrte. [2]	63(0x3F)	frD	///		frB	///	1 1	0 1 0 1	A	**frsqrte.**
fmsub	63(0x3F)	frD	frA		frB	frC	1 1	1 0 0 0	A	**fmsub**
fmsub.	63(0x3F)	frD	frA		frB	frC	1 1	1 0 0 1	A	**fmsub.**
fmadd	63(0x3F)	frD	frA		frB	frC	1 1	1 0 1 0	A	**fmadd**
fmadd.	63(0x3F)	frD	frA		frB	frC	1 1	1 0 1 1	A	**fmadd.**
fnmsub	63(0x3F)	frD	frA		frB	frC	1 1	1 1 0 0	A	**fnmsub**
fnmsub.	63(0x3F)	frD	frA		frB	frC	1 1	1 1 0 1	A	**fnmsub.**
fnmadd	63(0x3F)	frD	frA		frB	frC	1 1	1 1 1 0	A	**fnmadd**
fnmadd.	63(0x3F)	frD	frA		frB	frC	1 1	1 1 1 1	A	**fnmadd.**
fcmpo	63(0x3F)	crfD //	frA		frB	0 0 0	0 1 0 0	0 0 0 /	X	**fcmpo**
mtfsb1	63(0x3F)	crbD	///			0 0 0	0 1 0 0	1 1 0 0	X	**mtfsb1**
mtfsb1.	63(0x3F)	crbD	///			0 0 0	0 1 0 0	1 1 0 1	X	**mtfsb1.**
fneg	63(0x3F)	frD	///		frB	0 0 0	0 1 0 1	0 0 0 0	X	**fneg**
fneg.	63(0x3F)	frD	///		frB	0 0 0	0 1 0 1	0 0 0 1	X	**fneg.**
mcrfs	63(0x3F)	crfD //	crfS	///		0 0 0	1 0 0 0	0 0 0 /	X	**mcrfs**
mtfsb0	63(0x3F)	crbD	///			0 0 0	1 0 0 0	1 1 0 0	X	**mtfsb0**
mtfsb0.	63(0x3F)	crbD	///			0 0 0	1 0 0 0	1 1 0 1	X	**mtfsb0.**
fmr	63(0x3F)	frD	///		frB	0 0 0	1 0 0 1	0 0 0 0	A	**fmr**
fmr.	63(0x3F)	frD	///		frB	0 0 0	1 0 0 1	0 0 0 1	A	**fmr.**

Table A-2. Instructions Sorted by Primary Opcodes (Decimal and Hexadecimal) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20	21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
mtfsfi	63(0x3F)	crfD	///		IMM	/	0 0 1	0 0 0 0	1 1 0 0	X	**mtfsfi**
mtfsfi.	63(0x3F)	crfD	///		IMM	/	0 0 1	0 0 0 0	1 1 0 1	X	**mtfsfi.**
fnabs	63(0x3F)	frD	///		frB		0 0 1	0 0 0 1	0 0 0 0	X	**fnabs**
fnabs.	63(0x3F)	frD	///		frB		0 0 1	0 0 0 1	0 0 0 1	X	**fnabs.**
fabs	63(0x3F)	frD	///		frB		0 1 0	0 0 0 1	0 0 0 0	X	**fabs**
fabs.	63(0x3F)	frD	///		frB		0 1 0	0 0 0 1	0 0 0 1	X	**fabs.**
mffs	63(0x3F)	frD	///				1 0 0	1 0 0 0	1 1 1 0	X	**mffs**
mffs.	63(0x3F)	frD	///				1 0 0	1 0 0 0	1 1 1 1	X	**mffs.**
mtfsf	63(0x3F)	/	FM	/	frB		1 0 1	1 0 0 0	1 1 1 0	XFL	**mtfsf**
mtfsf.	63(0x3F)	/	FM	/	frB		1 0 1	1 0 0 0	1 1 1 1	XFL	**mtfsf.**

1 Supervisor-level instruction

2 Optional to the PowerPC classic architecture

3 Access level is detemined by whether the SPR is defined as a user or supervisor level SPR.

A.3 Instructions Sorted by Mnemonic (Binary)

Table A-3 lists instructions in alphabetical order by mnemonic with binary values. This list also includes simplified mnemonics and their equivalents using standard mnemonics.

Table A-3. Instructions Sorted by Mnemonic (Binary)

Mnemonic	0 1 2 3	4 5	6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
add	0 1 1 1	1 1	rD		rA	rB	0 1 0	0 0 0 1	0 1 0 0	X	**add**
add.	0 1 1 1	1 1	rD		rA	rB	0 1 0	0 0 0 1	0 1 0 1	X	**add.**
addc	0 1 1 1	1 1	rD		rA	rB	0 0 0	0 0 0 1	0 1 0 0	X	**addc**
addc.	0 1 1 1	1 1	rD		rA	rB	0 0 0	0 0 0 1	0 1 0 1	X	**addc.**
addco	0 1 1 1	1 1	rD		rA	rB	1 0 0	0 0 0 1	0 1 0 0	X	**addco**
addco.	0 1 1 1	1 1	rD		rA	rB	1 0 0	0 0 0 1	0 1 0 1	X	**addco.**
adde	0 1 1 1	1 1	rD		rA	rB	0 0 1	0 0 0 1	0 1 0 0	X	**adde**
adde.	0 1 1 1	1 1	rD		rA	rB	0 0 1	0 0 0 1	0 1 0 1	X	**adde.**
addeo	0 1 1 1	1 1	rD		rA	rB	1 0 1	0 0 0 1	0 1 0 0	X	**addeo**
addeo.	0 1 1 1	1 1	rD		rA	rB	1 0 1	0 0 0 1	0 1 0 1	X	**addeo.**
addi	0 0 1 1	1 0	rD		rA	SIMM				D	**addi**
addic	0 0 1 1	0 0	rD		rA	SIMM				D	**addic**
addic.	0 0 1 1	0 1	rD		rA	SIMM				D	**addic.**
addis	0 0 1 1	1 1	rD		rA	SIMM				D	**addis**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
addme	0	1	1	1	1	1	rD						rA				///				0	0	1		1	1	0	1	0	1	0	0	X	**addme**
addme.	0	1	1	1	1	1	rD						rA				///				0	0	1		1	1	0	1	0	1	0	1	X	**addme.**
addmeo	0	1	1	1	1	1	rD						rA				///				1	0	1		1	1	0	1	0	1	0	0	X	**addmeo**
addmeo.	0	1	1	1	1	1	rD						rA				///				1	0	1		1	1	0	1	0	1	0	1	X	**addmeo.**
addo	0	1	1	1	1	1	rD						rA				rB				1	1	0		0	0	0	1	0	1	0	0	X	**addo**
addo.	0	1	1	1	1	1	rD						rA				rB				1	1	0		0	0	0	1	0	1	0	1	X	**addo.**
addze	0	1	1	1	1	1	rD						rA				///				0	0	1		1	0	0	1	0	1	0	0	X	**addze**
addze.	0	1	1	1	1	1	rD						rA				///				0	0	1		1	0	0	1	0	1	0	1	X	**addze.**
addzeo	0	1	1	1	1	1	rD						rA				///				1	0	1		1	0	0	1	0	1	0	0	X	**addzeo**
addzeo.	0	1	1	1	1	1	rD						rA				///				1	0	1		1	0	0	1	0	1	0	1	X	**addzeo.**
and	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	0	1	1	1	0	0	0	X	**and**
and.	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	0	1	1	1	0	0	1	X	**and.**
andc	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	1	1	1	1	0	0	0	X	**andc**
andc.	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	1	1	1	1	0	0	1	X	**andc.**
andi.	0	1	1	1	0	0	rS						rA				UIMM																D	**andi.**
andis.	0	1	1	1	0	1	rS						rA				UIMM																D	**andis.**
b	0	1	0	0	1	0	LI																								0	0	I	**b**
ba	0	1	0	0	1	0	LI																								1	0	I	**ba**
bc	0	1	0	0	0	0	BO						BI				BD														0	0	B	**bc**
bca	0	1	0	0	0	0	BO						BI				BD														1	0	B	**bca**
bcctr	0	1	0	0	1	1	BO						BI				///				1	0	0		0	0	1	0	0	0	0	0	XL	**bcctr**
bcctrl	0	1	0	0	1	1	BO						BI				///				1	0	0		0	0	1	0	0	0	0	1	XL	**bcctrl**
bcl	0	1	0	0	0	0	BO						BI				BD														0	1	B	**bcl**
bcla	0	1	0	0	0	0	BO						BI				BD														1	1	B	**bcla**
bclr	0	1	0	0	1	1	BO						BI				///				0	0	0		0	0	1	0	0	0	0	0	XL	**bclr**
bclrl	0	1	0	0	1	1	BO						BI				///				0	0	0		0	0	1	0	0	0	0	1	XL	**bclrl**

bctr	**bctr** [1]	equivalent to	**bcctr 20,0**	**bctr**
bctrl	**bctrl** [1]	equivalent to	**bcctrl 20,0**	**bctrl**
bdnz	**bdnz target** [1]	equivalent to	**bc 16,0,**target	**bdnz**
bdnza	**bdnza** target [1]	equivalent to	**bca 16,0,**target	**bdnza**
bdnzf	**bdnzf BI,**target	equivalent to	**bc 0,BI,**target	**bdnzf**
bdnzfa	**bdnzfa BI,**target	equivalent to	**bca 0,BI,**target	**bdnzfa**
bdnzfl	**bdnzfl BI,**target	equivalent to	**bcl 0,BI,**target	**bdnzfl**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
bdnzfla	**bdnzfla BI,**target		equivalent to	**bcla 0,BI,**target						**bdnzfla**
bdnzflr	**bdnzflr BI**		equivalent to	**bclr 0,BI**						**bdnzflr**
bdnzflrl	**bdnzflrl BI**		equivalent to	**bclrl 0,BI**						**bdnzflrl**
bdnzl	**bdnzl** target [1]		equivalent to	**bcl 16,0,**target						**bdnzl**
bdnzla	**bdnzla** target [1]		equivalent to	**bcla 16,0,**target						**bdnzla**
bdnzlr	**bdnzlr BI**		equivalent to	**bclr 16,BI**						**bdnzlr**
bdnzlrl	**bdnzlrl** [1]		equivalent to	**bclrl 16,0**						**bdnzlrl**
bdnzt	**bdnzt BI,**target		equivalent to	**bc 8,BI,**target						**bdnzt**
bdnzta	**bdnzta BI,**target		equivalent to	**bca 8,BI,**target						**bdnzta**
bdnztl	**bdnztl BI,**target		equivalent to	**bcl 8,0,**target						**bdnztl**
bdnztla	**bdnztla BI,**target		equivalent to	**bcla 8,BI,**target						**bdnztla**
bdnztlr	**bdnztlr BI**		equivalent to	**bclr 8,BI**						**bdnztlr**
bdnztlr	**bdnztlr BI**		equivalent to	**bclr 8,BI**						**bdnztlr**
bdnztlrl	**bdnztlrl BI**		equivalent to	**bclrl 8,BI**						**bdnztlrl**
bdz	**bdz** target [1]		equivalent to	**bc 18,0,**target						**bdz**
bdza	**bdza** target [1]		equivalent to	**bca 18,0,**target						**bdza**
bdzf	**bdzf BI,**target		equivalent to	**bc 2,BI,**target						**bdzf**
bdzfa	**bdzfa BI,**target		equivalent to	**bca 2,BI,**target						**bdzfa**
bdzfl	**bdzfl BI,**target		equivalent to	**bcl 2,BI,**target						**bdzfl**
bdzfla	**bdzfla BI,**target		equivalent to	**bcla 2,BI,**target						**bdzfla**
bdzflr	**bdzflr BI**		equivalent to	**bclr 2,BI**						**bdzflr**
bdzflrl	**bdzflrl BI**		equivalent to	**bclrl 2,BI**						**bdzflrl**
bdzl	**bdzl** target [1]		equivalent to	**bcl 18,BI,**target						**bdzl**
bdzla	**bdzla** target [1]		equivalent to	**bcla 18,BI,**target						**bdzla**
bdzlr	**bdzlr** [1]		equivalent to	**bclr 18,0**						**bdzlr**
bdzlrl	**bdzlrl** [1]		equivalent to	**bclrl 18,0**						**bdzlrl**
bdzt	**bdzt BI,**target		equivalent to	**bc 10,BI,**target						**bdzt**
bdzta	**bdzta BI,**target		equivalent to	**bca 10,BI,**target						**bdzta**
bdztl	**bdztl BI,**target		equivalent to	**bcl 10,BI,**target						**bdztl**
bdztla	**bdztla BI,**target		equivalent to	**bcla 10,BI,**target						**bdztla**
bdztlrl	**bdztlrl BI**		equivalent to	**bclrl 10, BI**						**bdztlrl**
beq	**beq cr**S,target		equivalent to	**bc 12,**BI[2],target						**beq**
beqa	**beqa cr**S,target		equivalent to	**bca 12,**BI[2],target						**beqa**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
beqctr	**beqctr crS**,target		equivalent to	**bcctr 12,**BI[2],target						**beqctr**
beqctrl	**beqctrl crS**,target		equivalent to	**bcctrl 12,**BI[2],target						**beqctrl**
beql	**beql crS**,target		equivalent to	**bcl 12,**BI[2],target						**beql**
beqla	**beqla crS**,target		equivalent to	**bcla 12,**BI[2],target						**beqla**
beqlr	**beqlr crS**,target		equivalent to	**bclr 12,**BI[2],target						**beqlr**
beqlrl	**beqlrl crS**,target		equivalent to	**bclrl 12,**BI[2],target						**beqlrl**
bf	**bf BI,**target		equivalent to	**bc 4,BI,**target						**bf**
bfa	**bfa BI,**target		equivalent to	**bca 4,BI,**target						**bfa**
bfctr	**bfctr BI**		equivalent to	**bcctr 4,BI**						**bfctr**
bfctrl	**bfctrl BI**		equivalent to	**bcctrl 4,BI**						**bfctrl**
bfl	**bfl BI,**target		equivalent to	**bcl 4,BI,**target						**bfl**
bfla	**bfla BI,**target		equivalent to	**bcla 4,BI,**target						**bfla**
bflr	**bflr BI**		equivalent to	**bclr 4,BI**						**bflr**
bflrl	**bflrl BI**		equivalent to	**bclrl 4,BI**						**bflrl**
bge	**bge crS**,target		equivalent to	**bc 4,**BI[3],target						**bge**
bgea	**bgea crS**,target		equivalent to	**bca 4,**BI[3],target						**bgea**
bgectr	**bgectr crS**,target		equivalent to	**bcctr 4,**BI[3],target						**bgectr**
bgectrl	**bgectrl crS**,target		equivalent to	**bcctrl 4,**BI[3],target						**bgectrl**
bgel	**bgel crS**,target		equivalent to	**bcl 4,**BI[3],target						**bgel**
bgela	**bgela crS**,target		equivalent to	**bcla 4,**BI[3],target						**bgela**
bgelr	**bgelr crS**,target		equivalent to	**bclr 4,**BI[3],target						**bgelr**
bgelrl	**bgelrl crS**,target		equivalent to	**bclrl 4,**BI[3],target						**bgelrl**
bgt	**bgt crS**,target		equivalent to	**bc 12,**BI[4],target						**bgt**
bgta	**bgta crS**,target		equivalent to	**bca 12,**BI[4],target						**bgta**
bgtctr	**bgtctr crS**,target		equivalent to	**bcctr 12,**BI[4],target						**bgtctr**
bgtctrl	**bgtctrl crS**,target		equivalent to	**bcctrl 12,**BI[4],target						**bgtctrl**
bgtl	**bgtl crS**,target		equivalent to	**bcl 12,**BI[4],target						**bgtl**
bgtla	**bgtla crS**,target		equivalent to	**bcla 12,**BI[4],target						**bgtla**
bgtlr	**bgtlr crS**,target		equivalent to	**bclr 12,**BI[4],target						**bgtlr**
bgtlrl	**bgtlrl crS**,target		equivalent to	**bclrl 12,**BI[4],target						**bgtlrl**
bl	0 1 0 0	1 0	LI					0 1	I	**bl**
bla	0 1 0 0	1 0	LI					1 1	I	**bla**
ble	**ble crS**,target		equivalent to	**bc 4,**BI[4],target						**ble**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
blea	**blea crS**,target		equivalent to	**bca 4,**BI[4],target						**blea**
blectr	**blectr crS**,target		equivalent to	**bcctr 4,**BI[4],target						**blectr**
blectrl	**blectrl crS**,target		equivalent to	**bcctrl 4,**BI[4],target						**blectrl**
blel	**blel crS**,target		equivalent to	**bcl 4,**BI[4],target						**blel**
blela	**blela crS**,target		equivalent to	**bcla 4,**BI[4],target						**blela**
blelr	**blelr crS**,target		equivalent to	**bclr 4,**BI[4],target						**blelr**
blelrl	**blelrl crS**,target		equivalent to	**bclrl 4,**BI[4],target						**blelrl**
blr	**blr** [1]		equivalent to	**bclr 20,0**						**blr**
blrl	**blrl** [1]		equivalent to	**bclrl 20,0**						**blrl**
blt	**blt crS**,target		equivalent to	**bc 12,**BI,target						**blt**
blta	**blta crS**,target		equivalent to	**bca 12,**BI[3],target						**blta**
bltctr	**bltctr crS**,target		equivalent to	**bcctr 12,**BI[3],target						**bltctr**
bltctrl	**bltctrl crS**,target		equivalent to	**bcctrl 12,**BI[3],target						**bltctrl**
bltl	**bltl crS**,target		equivalent to	**bcl 12,**BI[3],target						**bltl**
bltla	**bltla crS**,target		equivalent to	**bcla 12,**BI[3],target						**bltla**
bltlr	**bltlr crS**,target		equivalent to	**bclr 12,**BI[3],target						**bltlr**
bltlrl	**bltlrl crS**,target		equivalent to	**bclrl 12,**BI[3],target						**bltlrl**
bne	**bne crS**,target		equivalent to	**bc 4,**BI[3],target						**bne**
bnea	**bnea crS**,target		equivalent to	**bca 4,**BI[3],target						**bnea**
bnectr	**bnectr crS**,target		equivalent to	**bcctr 4,**BI[3],target						**bnectr**
bnectrl	**bnectrl crS**,target		equivalent to	**bcctrl 4,**BI[3],target						**bnectrl**
bnel	**bnel crS**,target		equivalent to	**bcl 4,**BI[3],target						**bnel**
bnela	**bnela crS**,target		equivalent to	**bcla 4,**BI[3],target						**bnela**
bnelr	**bnelr crS**,target		equivalent to	**bclr 4,**BI[3],target						**bnelr**
bnelrl	**bnelrl crS**,target		equivalent to	**bclrl 4,**BI[3],target						**bnelrl**
bng	**bng crS**,target		equivalent to	**bc 4,**BI[4],target						**bng**
bnga	**bnga crS**,target		equivalent to	**bca 4,**BI[4],target						**bnga**
bngctr	**bngctr crS**,target		equivalent to	**bcctr 4,**BI[4],target						**bngctr**
bngctrl	**bngctrl crS**,target		equivalent to	**bcctrl 4,**BI[4],target						**bngctrl**
bngl	**bngl crS**,target		equivalent to	**bcl 4,**BI[4],target						**bngl**
bngla	**bngla crS**,target		equivalent to	**bcla 4,**BI[4],target						**bngla**
bnglr	**bnglr crS**,target		equivalent to	**bclr 4,**BI[4],target						**bnglr**
bnglrl	**bnglrl crS**,target		equivalent to	**bclrl 4,**BI[4],target						**bnglrl**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
bnl	**bnl cr**S,target		equivalent to	**bc 4,**BI[3],target						**bnl**
bnla	**bnla cr**S,target		equivalent to	**bca 4,**BI[3],target						**bnla**
bnlctr	**bnlctr cr**S,target		equivalent to	**bcctr 4,**BI[3],target						**bnlctr**
bnlctrl	**bnlctrl cr**S,target		equivalent to	**bcctrl 4,**BI[3],target						**bnlctrl**
bnll	**bnll cr**S,target		equivalent to	**bcl 4,**BI[3],target						**bnll**
bnlla	**bnlla cr**S,target		equivalent to	**bcla 4,**BI[3],target						**bnlla**
bnllr	**bnllr cr**S,target		equivalent to	**bclr 4,**BI[3],target						**bnllr**
bnllrl	**bnllrl cr**S,target		equivalent to	**bclrl 4,**BI[3],target						**bnllrl**
bns	**bns cr**S,target		equivalent to	**bc 4,**BI[5],target						**bns**
bnsa	**bnsa cr**S,target		equivalent to	**bca 4,**BI[5],target						**bnsa**
bnsctr	**bnsctr cr**S,target		equivalent to	**bcctr 4,**BI[5],target						**bnsctr**
bnsctrl	**bnsctrl cr**S,target		equivalent to	**bcctrl 4,**BI[5],target						**bnsctrl**
bnsl	**bnsl cr**S,target		equivalent to	**bcl 4,**BI[5],target						**bnsl**
bnsla	**bnsla cr**S,target		equivalent to	**bcla 4,**BI[5],target						**bnsla**
bnslr	**bnslr cr**S,target		equivalent to	**bclr 4,**BI[5],target						**bnslr**
bnslrl	**bnslrl cr**S,target		equivalent to	**bclrl 4,**BI[5],target						**bnslrl**
bnu	**bnu cr**S,target		equivalent to	**bc 4,**BI[5],target						**bnu**
bnua	**bnua cr**S,target		equivalent to	**bca 4,**BI[5],target						**bnua**
bnuctr	**bnuctr cr**S,target		equivalent to	**bcctr 4,**BI[5],target						**bnuctr**
bnuctrl	**bnuctrl cr**S,target		equivalent to	**bcctrl 4,**BI[5],target						**bnuctrl**
bnul	**bnul cr**S,target		equivalent to	**bcl 4,**BI[5],target						**bnul**
bnula	**bnula cr**S,target		equivalent to	**bcla 4,**BI[5],target						**bnula**
bnulr	**bnulr cr**S,target		equivalent to	**bclr 4,**BI[5],target						**bnulr**
bnulrl	**bnulrl cr**S,target		equivalent to	**bclrl 4,**BI[5],target						**bnulrl**
bso	**bso cr**S,target		equivalent to	**bc 12,**BI[5],target						**bso**
bsoa	**bsoa cr**S,target		equivalent to	**bca 12,**BI[5],target						**bsoa**
bsoctr	**bsoctr cr**S,target		equivalent to	**bcctr 12,**BI[5],target						**bsoctr**
bsoctrl	**bsoctrl cr**S,target		equivalent to	**bcctrl 12,**BI[5],target						**bsoctrl**
bsol	**bsol cr**S,target		equivalent to	**bcl 12,**BI[5],target						**bsol**
bsola	**bsola cr**S,target		equivalent to	**bcla 12,**BI[5],target						**bsola**
bsolr	**bsolr cr**S,target		equivalent to	**bclr 12,**BI[5],target						**bsolr**
bsolrl	**bsolrl cr**S,target		equivalent to	**bclrl 12,**BI[5],target						**bsolrl**
bt	**bt BI,**target		equivalent to	**bc 12,BI,**target						**bt**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
bta	**bta BI**,target		equivalent to	**bca 12,BI**,target						**bta**
btctr	**btctr BI**		equivalent to	**bcctr 12,BI**						**btctr**
btctrl	**btctrl BI**		equivalent to	**bcctrl 12,BI**						**btctrl**
btl	**btl BI**,target		equivalent to	**bcl 12,BI**,target						**btl**
btla	**btla BI**,target		equivalent to	**bcla 12,BI**,target						**btla**
btlr	**btlr BI**		equivalent to	**bclr 12,BI**						**btlr**
btlrl	**btlrl BI**		equivalent to	**bclrl 12,BI**						**btlrl**
bun	**bun cr**S,target		equivalent to	**bc 12,**BI[5],target						**bun**
buna	**buna cr**S,target		equivalent to	**bca 12,**BI[5],target						**buna**
bunctr	**bunctr cr**S,target		equivalent to	**bcctr 12,**BI[5],target						**bunctr**
bunctrl	**bunctrl cr**S,target		equivalent to	**bcctrl 12,**BI[5],target						**bunctrl**
bunl	**bunl cr**S,target		equivalent to	**bcl 12,**BI[5],target						**bunl**
bunla	**bunla cr**S,target		equivalent to	**bcla 12,**BI[5],target						**bunla**
bunlr	**bunlr cr**S,target		equivalent to	**bclr 12,**BI[5],target						**bunlr**
bunlrl	**bunlrl cr**S,target		equivalent to	**bclrl 12,**BI[5],target						**bunlrl**
clrlslwi	**clrlslwi r**A,**r**S,*b*,*n* ($n \le b \le 31$)		equivalent to	**rlwinm r**A,**r**S,*n*,*b* − *n*,31 − *n*						**clrlslwi**
clrlwi	**clrlwi r**A,**r**S,*n* (n < 32)		equivalent to	**rlwinm r**A,**r**S,**0**,*n*,**31**						**clrlwi**
clrrwi	**clrrwi r**A,**r**S,*n* (n < 32)		equivalent to	**rlwinm r**A,**r**S,**0,0,31** − *n*						**clrrwi**
cmp	0 1 1 1	1 1	crfD / L	rA	rB	0 0 0	0 0 0 0	0 0 0 /	X	**cmp**
cmpi	0 0 1 0	1 1	crfD / L	rA	SIMM				D	**cmpi**
cmpl	0 1 1 1	1 1	/ L	rA	rB	///	0 0 0	0 1 0 0 0 0 0 /	X	**cmpl**
cmpli	0 0 1 0	1 0	crfD / L	rA	UIMM				D	**cmpli**
cmplw	**cmplw cr**D,**r**A,**r**B		equivalent to	**cmpl cr**D,**0**,**r**A,**r**B						**cmplw**
cmplwi	**cmplwi cr**D,**r**A,**UIMM**		equivalent to	**cmpli cr**D,**0**,**r**A,UIMM						**cmplwi**
cmpw	**cmpw cr**D,**r**A,**r**B		equivalent to	**cmp cr**D,**0**,**r**A,**r**B						**cmpw**
cmpwi	**cmpwi cr**D,**r**A,**SIMM**		equivalent to	**cmpi cr**D,**0**,**r**A,SIMM						**cmpwi**
cntlzw	0 1 1 1	1 1	rS	rA	///	0 0 0	0 0 1 1	0 1 0 0	X	**cntlzw**
cntlzw.	0 1 1 1	1 1	rS	rA	///	0 0 0	0 0 1 1	0 1 0 1	X	**cntlzw.**
crand	0 1 0 0	1 1	crbD	crbA	crbB	0 1 0	0 0 0 0	0 0 1 /	XL	**crand**
crandc	0 1 0 0	1 1	crbD	crbA	crbB	0 0 1	0 0 0 0	0 0 1 /	XL	**crandc**
crclr	**crclr bx**		equivalent to	**crxor** bx,bx,bx						**crclr**
creqv	0 1 0 0	1 1	crbD	crbA	crbB	0 1 0	0 1 0 0	0 0 1 /	XL	**creqv**
crmove	**crmove bx,by**		equivalent to	**cror** bx,by,by						**crmove**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
crnand	0	1	0	0	1	1	crbD						crbA				crbB				0	0	1		1	1	0	0	0	0	1	/	XL	**crnand**
crnor	0	1	0	0	1	1	crbD						crbA				crbB				0	0	0		0	1	0	0	0	0	1	/	XL	**crnor**
crnot	**crnot bx,by**												equivalent to				**crnor** bx,by,by																	**crnot**
cror	0	1	0	0	1	1	crbD						crbA				crbB				0	1	1		1	0	0	0	0	0	1	/	XL	**cror**
crorc	0	1	0	0	1	1	crbD						crbA				crbB				0	1	1		0	1	0	0	0	0	1	/	XL	**crorc**
crset	**crset bx**												equivalent to				**creqv** bx,bx,bx																	**crset**
crxor	0	1	0	0	1	1	crbD						crbA				crbB				0	0	1		1	0	0	0	0	0	1	/	XL	**crxor**
dcba [6]	0	1	1	1	1	1	///						rA				rB				1	0	1		1	1	1	0	1	1	0	/	X	**dcba**
dcbf	0	1	1	1	1	1	///						rA				rB				0	0	0		1	0	1	0	1	1	0	/	X	**dcbf**
dcbi [7]	0	1	1	1	1	1	///						rA				rB				0	1	1		1	0	1	0	1	1	0	/	X	**dcbi**
dcbst	0	1	1	1	1	1	///						rA				rB				0	0	0		0	1	1	0	1	1	0	/	X	**dcbst**
dcbt	0	1	1	1	1	1	CT						rA				rB				0	1	0		0	0	1	0	1	1	0	/	X	**dcbt**
dcbtst	0	1	1	1	1	1	CT						rA				rB				0	0	1		1	1	1	0	1	1	0	/	X	**dcbtst**
dcbz	0	1	1	1	1	1	///						rA				rB				1	1	1		1	1	1	0	1	1	0	/	X	**dcbz**
divw	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	1	0	1	0	1	1	0	X	**divw**
divw.	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	1	0	1	0	1	1	1	X	**divw.**
divwo	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	1	0	1	0	1	1	0	X	**divwo**
divwo.	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	1	0	1	0	1	1	1	X	**divwo.**
divwu	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	0	0	1	0	1	1	0	X	**divwu**
divwu.	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	0	0	1	0	1	1	1	X	**divwu.**
divwuo	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	0	0	1	0	1	1	0	X	**divwuo**
divwuo.	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	0	0	1	0	1	1	1	X	**divwuo.**
dss	**dss STRM**												equivalent to				**dss** STRM,**0**																	**dss**
eciwx [6]	0	1	1	1	1	1	rD						rA				rB				0	1	0		0	1	1	0	1	1	0	0	X	**eciwx**
ecowx [6]	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	1	1	0	1	1	0	0	X	**ecowx**
eieio	0	1	1	1	1	1	///						///				///				1	1	0		1	0	1	0	1	1	0	0	X	**eieio**
eqv	0	1	1	1	1	1	rD						rA				rB				0	1	0		0	0	1	1	1	0	0	0	X	**eqv**
eqv.	0	1	1	1	1	1	rD						rA				rB				0	1	0		0	0	1	1	1	0	0	1	X	**eqv.**
extlwi	**extlwi** **r**A,**r**S,*n*,*b* (n > 0)												equivalent to				**rlwinm** **r**A,**r**S,*b*,**0**,*n* – 1																	**extlwi**
extrwi	**extrwi** **r**A,**r**S,*n*,*b* (*n* > 0)												equivalent to				**rlwinm** **r**A,**r**S,*b* + *n*, 32 – *n*,**31**																	**extrwi**
extsb	0	1	1	1	1	1	rS						rA				///				1	1	1		0	1	1	1	0	1	0	0	X	**extsb**
extsb.	0	1	1	1	1	1	rS						rA				///				1	1	1		0	1	1	1	0	1	0	1	X	**extsb.**
extsh	0	1	1	1	1	1	rS						rA				///				1	1	1		0	0	1	1	0	1	0	0	X	**extsh**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
extsh.	0	1	1	1	1	1	rS						rA				///				1	1	1		0	0	1	1	0	1	0	1	X	**extsh.**
fabs	1	1	1	1	1	1	frD						///				frB				0	1	0		0	0	0	1	0	0	0	0	X	**fabs**
fabs.	1	1	1	1	1	1	frD						///				frB				0	1	0		0	0	0	1	0	0	0	1	X	**fabs.**
fadd	1	1	1	1	1	1	frD						frA				frB				///						1	0	1	0	1	0	A	**fadd**
fadd.	1	1	1	1	1	1	frD						frA				frB				///						1	0	1	0	1	1	A	**fadd.**
fadds	1	1	1	0	1	1	frD						frA				frB				///						1	0	1	0	1	0	A	**fadds**
fadds.	1	1	1	0	1	1	frD						frA				frB				///						1	0	1	0	1	1	A	**fadds.**
fcmpo	1	1	1	1	1	1	crfD			//			frA				frB				0	0	0		0	1	0	0	0	0	0	/	X	**fcmpo**
fcmpu	1	1	1	1	1	1	crfD			//			frA				frB				0	0	0		0	0	0	0	0	0	0	/	X	**fcmpu**
fctiw	1	1	1	1	1	1	frD						///				frB				0	0	0		0	0	0	1	1	1	0	0	X	**fctiw**
fctiw.	1	1	1	1	1	1	frD						///				frB				0	0	0		0	0	0	1	1	1	0	1	X	**fctiw.**
fctiwz	1	1	1	1	1	1	frD						///				frB				0	0	0		0	0	0	1	1	1	1	0	X	**fctiwz**
fctiwz.	1	1	1	1	1	1	frD						///				frB				0	0	0		0	0	0	1	1	1	1	1	X	**fctiwz.**
fdiv	1	1	1	1	1	1	frD						frA				frB				///						1	0	0	1	0	0	A	**fdiv**
fdiv.	1	1	1	1	1	1	frD						frA				frB				///						1	0	0	1	0	1	A	**fdiv.**
fdivs	1	1	1	0	1	1	frD						frA				frB				///						1	0	0	1	0	0	A	**fdivs**
fdivs.	1	1	1	0	1	1	frD						frA				frB				///						1	0	0	1	0	1	A	**fdivs.**
fmadd	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	1	0	A	**fmadd**
fmadd.	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	1	1	A	**fmadd.**
fmadds	1	1	1	0	1	1	frD						frA				frB				frC						1	1	1	0	1	0	A	**fmadds**
fmadds.	1	1	1	0	1	1	frD						frA				frB				frC						1	1	1	0	1	1	A	**fmadds.**
fmr	1	1	1	1	1	1	frD						///				frB				0	0	0		1	0	0	1	0	0	0	0	A	**fmr**
fmr.	1	1	1	1	1	1	frD						///				frB				0	0	0		1	0	0	1	0	0	0	1	A	**fmr.**
fmsub	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	0	0	A	**fmsub**
fmsub.	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	0	1	A	**fmsub.**
fmsubs	1	1	1	0	1	1	frD						frA				frB				frC						1	1	1	0	0	0	A	**fmsubs**
fmsubs.	1	1	1	0	1	1	frD						frA				frB				frC						1	1	1	0	0	1	A	**fmsubs.**
fmul	1	1	1	1	1	1	frD						frA				///				frC						1	1	0	0	1	0	A	**fmul**
fmul.	1	1	1	1	1	1	frD						frA				///				frC						1	1	0	0	1	1	A	**fmul.**
fmuls	1	1	1	0	1	1	frD						frA				///				frC						1	1	0	0	1	0	A	**fmuls**
fmuls.	1	1	1	0	1	1	frD						frA				///				frC						1	1	0	0	1	1	A	**fmuls.**
fnabs	1	1	1	1	1	1	frD						///				frB				0	0	1		0	0	0	1	0	0	0	0	X	**fnabs**
fnabs.	1	1	1	1	1	1	frD						///				frB				0	0	1		0	0	0	1	0	0	0	1	X	**fnabs.**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
fneg	1	1	1	1	1	1	frD						///				frB				0	0	0		0	1	0	1	0	0	0	0	X	**fneg**
fneg.	1	1	1	1	1	1	frD						///				frB				0	0	0		0	1	0	1	0	0	0	1	X	**fneg.**
fnmadd	1	1	1	1	1	1	frD						frA				frB				frC				1	1	1	1	1	1	1	0	A	**fnmadd**
fnmadd.	1	1	1	1	1	1	frD						frA				frB				frC				1	1	1	1	1	1	1	1	A	**fnmadd.**
fnmadds	1	1	1	0	1	1	frD						frA				frB				frC				1	1	1	1	1	1	1	0	A	**fnmadds**
fnmadds.	1	1	1	0	1	1	frD						frA				frB				frC				1	1	1	1	1	1	1	1	A	**fnmadds.**
fnmsub	1	1	1	1	1	1	frD						frA				frB				frC				1	1	1	1	1	1	0	0	A	**fnmsub**
fnmsub.	1	1	1	1	1	1	frD						frA				frB				frC				1	1	1	1	1	1	0	1	A	**fnmsub.**
fnmsubs	1	1	1	0	1	1	frD						frA				frB				frC				1	1	1	1	1	1	0	0	A	**fnmsubs**
fnmsubs.	1	1	1	0	1	1	frD						frA				frB				frC				1	1	1	1	1	1	0	1	A	**fnmsubs.**
fres [6]	1	1	1	0	1	1	frD						///				frB				///				1	1	0	0	0	0	0	0	A	**fres**
fres. [6]	1	1	1	0	1	1	frD						///				frB				///				1	1	0	0	0	0	0	1	A	**fres.**
frsp	1	1	1	1	1	1	frD						///				frB				0	0	0		0	0	0	1	1	0	0	0	X	**frsp**
frsp.	1	1	1	1	1	1	frD						///				frB				0	0	0		0	0	0	1	1	0	0	1	X	**frsp.**
frsqrte [6]	1	1	1	1	1	1	frD						///				frB				///				1	1	0	1	0	0	0	0	A	**frsqrte**
frsqrte. [6]	1	1	1	1	1	1	frD						///				frB				///				1	1	0	1	0	0	0	1	A	**frsqrte.**
fsel [6]	1	1	1	1	1	1	frD						frA				frB				frC				1	0	1	1	1	0	0	0	A	**fsel**
fsel. [6]	1	1	1	1	1	1	frD						frA				frB				frC				1	0	1	1	1	1	0	1	A	**fsel.**
fsqrt [6]	1	1	1	1	1	1	frD						///				frB				///				1	0	1	1	0	0	0	0	A	**fsqrt**
fsqrt. [6]	1	1	1	1	1	1	frD						///				frB				///				1	0	1	1	0	0	0	1	A	**fsqrt.**
fsqrts [6]	1	1	1	0	1	1	frD						///				frB				///				1	0	1	1	0	0	0	0	A	**fsqrts**
fsqrts. [6]	1	1	1	0	1	1	frD						///				frB				///				1	0	1	1	0	0	0	1	A	**fsqrts.**
fsub	1	1	1	1	1	1	frD						frA				frB				///				1	0	1	0	0	0	0	0	A	**fsub**
fsub.	1	1	1	1	1	1	frD						frA				frB				///				1	0	1	0	0	0	0	1	A	**fsub.**
fsubs	1	1	1	0	1	1	frD						frA				frB				///				1	0	1	0	0	0	0	0	A	**fsubs**
fsubs.	1	1	1	0	1	1	frD						frA				frB				///				1	0	1	0	0	0	0	1	A	**fsubs.**
icbi	0	1	1	1	1	1	///						rA				rB				1	1	1		1	0	1	0	1	1	0	/	X	**icbi**
inslwi	**inslwi rA,rS,*n*,*b* (*n* > 0)** equivalent to **rlwimi rA,rS,32 – *b*,*b*,(*b* + *n*) – 1**																																	**inslwi**
insrwi	**insrwi rA,rS,*n*,*b* (*n* > 0)** equivalent to **rlwimi rA,rS,32 – (*b* + *n*),*b*,(*b* + *n*) – 1**																																	**insrwi**
isync	0	1	0	0	1	1	///														0	0	1		0	0	1	0	1	1	0	/	XL	**isync**
la	**la rD,d(rA)** equivalent to **addi rD,rA,d**																																	**la**
lbz	1	0	0	0	1	0	rD						rA				D																D	**lbz**
lbzu	1	0	0	0	1	1	rD						rA				D																D	**lbzu**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
lbzux	0	1	1	1	1	1	rD						rA				rB				0	0	0		1	1	1	0	1	1	1	/	X	**lbzux**
lbzx	0	1	1	1	1	1	rD						rA				rB				0	0	0		1	0	1	0	1	1	1	/	X	**lbzx**
lfd	1	1	0	0	1	0	frD						rA				D																D	**lfd**
lfdu	1	1	0	0	1	1	frD						rA				D																D	**lfdu**
lfdux	0	1	1	1	1	1	frD						rA				rB				1	0	0		1	1	1	0	1	1	1	/	X	**lfdux**
lfdx	0	1	1	1	1	1	frD						rA				rB				1	0	0		1	0	1	0	1	1	1	/	X	**lfdx**
lfs	1	1	0	0	0	0	frD						rA				D																D	**lfs**
lfsu	1	1	0	0	0	1	frD						rA				D																D	**lfsu**
lfsux	0	1	1	1	1	1	frD						rA				rB				1	0	0		0	1	1	0	1	1	1	/	X	**lfsux**
lfsx	0	1	1	1	1	1	frD						rA				rB				1	0	0		0	0	1	0	1	1	1	/	X	**lfsx**
lha	1	0	1	0	1	0	rD						rA				D																D	**lha**
lhau	1	0	1	0	1	1	rD						rA				D																D	**lhau**
lhaux	0	1	1	1	1	1	rD						rA				rB				0	1	0		1	1	1	0	1	1	1	/	X	**lhaux**
lhax	0	1	1	1	1	1	rD						rA				rB				0	1	0		1	0	1	0	1	1	1	/	X	**lhax**
lhbrx	0	1	1	1	1	1	rD						rA				rB				1	1	0		0	0	1	0	1	1	0	/	X	**lhbrx**
lhz	1	0	1	0	0	0	rD						rA				D																D	**lhz**
lhzu	1	0	1	0	0	1	rD						rA				D																D	**lhzu**
lhzux	0	1	1	1	1	1	rD						rA				rB				0	1	0		0	1	1	0	1	1	1	/	X	**lhzux**
lhzx	0	1	1	1	1	1	rD						rA				rB				0	1	0		0	0	1	0	1	1	1	/	X	**lhzx**
li	**li rD,value**												equivalent to				**addi rD,0,**value																	**li**
lis	**lis rD,value**												equivalent to				**addis rD,0,**value																	**lis**
lmw	1	0	1	1	1	0	rD						rA				D																D	**lmw**
lswi	0	1	1	1	1	1	rD						rA				NB				1	0	0		1	0	1	0	1	0	1	/	X	**lswi**
lswx	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	1	0	1	0	1	/	X	**lswx**
lwarx	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	0	1	0	1	0	0	/	X	**lwarx**
lwbrx	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	1	0	1	1	0	/	X	**lwbrx**
lwz	1	0	0	0	0	0	rD						rA				D																D	**lwz**
lwzu	1	0	0	0	0	1	rD						rA				D																D	**lwzu**
lwzux	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	1	1	0	1	1	1	/	X	**lwzux**
lwzx	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	0	1	0	1	1	1	/	X	**lwzx**
mcrf	0	1	0	0	1	1	crfD			//			crfS				///				0	0	0		0	0	0	0	0	0	0	/	XL	**mcrf**
mcrfs	1	1	1	1	1	1	crfD			//			crfS				///				0	0	0		1	0	0	0	0	0	0	/	X	**mcrfs**
mcrxr	0	1	1	1	1	1	crfD			///											1	0	0		0	0	0	0	0	0	0	/	X	**mcrxr**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mfcr	**mtcr rS**												equivalent to				**mtcrf 0xFF,rS**																	**mfcr**
mfcr	0	1	1	1	1	1	rD						///								0	0	0	0	0	1	0	0	1	1	1	/	X	**mfcr**
mffs	1	1	1	1	1	1	frD						///								1	0	0	1	0	0	0	0	1	1	1	0	X	**mffs**
mffs.	1	1	1	1	1	1	frD						///								1	0	0	1	0	0	0	0	1	1	1	1	X	**mffs.**
mfmsr [7]	0	1	1	1	1	1	rD						///								0	0	0	1	0	1	0	0	1	1	1	/	X	**mfmsr**
mf*regname*	**mf*regname* rD**												equivalent to				**mfspr rD,SPR*n***																	**mf*regname***
mfspr [8]	0	1	1	1	1	1	rD						SPR[5–9]				SPR[0–4]				0	1	0	1	0	1	0	0	1	1	1	/	XFX	**mfspr**
mfsr [7]	0	1	1	1	1	1	rD						/	SR			///				1	0	0	1	0	1	0	0	1	1	1	0	X	**mfsr**
mfsrin [7]	0	1	1	1	1	1	rD						///				rB				1	0	1	0	0	1	0	0	1	1	1	0	X	**mfsrin**
mftb	0	1	1	1	1	1	rD						TBR[5–9]				TBR[0–4]				0	1	0	1	1	1	0	0	1	1	1	0	XFX	**mftb**
mr	**mr rA,rS**												equivalent to				**or rA,rS,rS**																	**mr**
mtcr	**mtcr rS**												equivalent to				**mtcrf** 0xFF,**rS**																	**mtcr**
mtcrf	0	1	1	1	1	1	rS						/	CRM						/	0	0	1	0	0	1	0	0	0	0	0	/	XFX	**mtcrf**
mtfsb0	1	1	1	1	1	1	crbD						///								0	0	0	1	0	0	0	1	1	0	0	0	X	**mtfsb0**
mtfsb0.	1	1	1	1	1	1	crbD						///								0	0	0	1	0	0	0	1	1	0	0	1	X	**mtfsb0.**
mtfsb1	1	1	1	1	1	1	crbD						///								0	0	0	0	1	0	0	1	1	0	0	0	X	**mtfsb1**
mtfsb1.	1	1	1	1	1	1	crbD						///								0	0	0	0	1	0	0	1	1	0	0	1	X	**mtfsb1.**
mtfsf	1	1	1	1	1	1	/	FM								/	frB				1	0	1	1	0	0	0	1	1	1	0	XFL	**mtfsf**
mtfsf.	1	1	1	1	1	1	/	FM								/	frB				1	0	1	1	0	0	0	1	1	1	1	XFL	**mtfsf.**
mtfsfi	1	1	1	1	1	1	crfD				///						IMM			/	0	0	1	0	0	0	0	1	1	0	0	X	**mtfsfi**
mtfsfi.	1	1	1	1	1	1	crfD				///						IMM			/	0	0	1	0	0	0	0	1	1	0	1	X	**mtfsfi.**
mtmsr [7]	0	1	1	1	1	1	rS						///								0	0	1	0	0	1	0	0	1	0	/	X	**mtmsr**
mt*regname*	**mt*regname* rS**												equivalent to				**mtspr SPR*n* rS**																	**mt*regname***
mtspr [8]	0	1	1	1	1	1	rS						SPR[5–9]				SPR[0–4]				0	1	1	1	0	1	0	0	1	1	1	/	XFX	**mtspr**
mtsr [7]	0	1	1	1	1	1	rS						/	SR			///				0	0	1	1	0	1	0	0	1	0	0	0	X	**mtsr**
mtsrin [7]	0	1	1	1	1	1	rS						///				rB				0	0	1	1	1	1	0	0	1	0	0	0	X	**mtsrin**
mulhw	0	1	1	1	1	1	rD						rA				rB			/	0	0	1	0	0	1	0	1	1	0	0	X	**mulhw**
mulhw.	0	1	1	1	1	1	rD						rA				rB			/	0	0	1	0	0	1	0	1	1	0	1	X	**mulhw.**
mulhwu	0	1	1	1	1	1	rD						rA				rB			/	0	0	0	0	0	1	0	1	1	0	0	X	**mulhwu**
mulhwu.	0	1	1	1	1	1	rD						rA				rB			/	0	0	0	0	0	1	0	1	1	0	1	X	**mulhwu.**
mulli	0	0	0	1	1	1	rD						rA				SIMM																D	**mulli**
mullw	0	1	1	1	1	1	rD						rA				rB			0	0	0	1	1	1	0	1	0	1	1	0	X	**mullw**
mullw.	0	1	1	1	1	1	rD						rA				rB			0	0	0	1	1	1	0	1	0	1	1	1	X	**mullw.**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mullwo	0	1	1	1	1	1	rD						rA				rB				1	0	1	1	1	0	1	0	1	1	0		X	**mullwo**
mullwo.	0	1	1	1	1	1	rD						rA				rB				1	0	1	1	1	0	1	0	1	1	1		X	**mullwo.**
nand	0	1	1	1	1	1	rS						rA				rB				0	1	1	1	0	1	1	1	0	0	0		X	**nand**
nand.	0	1	1	1	1	1	rS						rA				rB				0	1	1	1	0	1	1	1	0	0	1		X	**nand.**
neg	0	1	1	1	1	1	rD						rA				///				0	0	0	1	1	0	1	0	0	0	0		X	**neg**
neg.	0	1	1	1	1	1	rD						rA				///				0	0	0	1	1	0	1	0	0	0	1		X	**neg.**
nego	0	1	1	1	1	1	rD						rA				///				1	0	0	1	1	0	1	0	0	0	0		X	**nego**
nego.	0	1	1	1	1	1	rD						rA				///				1	0	0	1	1	0	1	0	0	0	1		X	**nego.**
nop	**nop**												equivalent to				**ori 0,0,0**																	**nop**
nor	0	1	1	1	1	1	rS						rA				rB				0	0	0	1	1	1	1	1	0	0	0		X	**nor**
nor.	0	1	1	1	1	1	rS						rA				rB				0	0	0	1	1	1	1	1	0	0	1		X	**nor.**
not	**not** rA,rS												equivalent to				**nor** rA,rS,rS																	**not**
or	0	1	1	1	1	1	rS						rA				rB				0	1	1	0	1	1	1	1	0	0	0		X	**or**
or.	0	1	1	1	1	1	rS						rA				rB				0	1	1	0	1	1	1	1	0	0	1		X	**or.**
orc	0	1	1	1	1	1	rS						rA				rB				0	1	1	0	0	1	1	1	0	0	0		X	**orc**
orc.	0	1	1	1	1	1	rS						rA				rB				0	1	1	0	0	1	1	1	0	0	1		X	**orc.**
ori	0	1	1	0	0	0	rS						rA				UIMM																D	**ori**
oris	0	1	1	0	0	1	rS						rA				UIMM																D	**oris**
rfi [7]	0	1	0	0	1	1	///														0	0	0	0	1	1	0	0	1	0	/		XL	**rfi**
rlwimi	0	1	0	1	0	0	rS						rA				SH				MB							ME				0	M	**rlwimi**
rlwimi.	0	1	0	1	0	0	rS						rA				SH				MB							ME				1	M	**rlwimi.**
rlwinm	0	1	0	1	0	1	rS						rA				SH				MB							ME				0	M	**rlwinm**
rlwinm.	0	1	0	1	0	1	rS						rA				SH				MB							ME				1	M	**rlwinm.**
rlwnm	0	1	0	1	1	1	rS						rA				rB				MB							ME				0	M	**rlwnm**
rlwnm.	0	1	0	1	1	1	rS						rA				rB				MB							ME				1	M	**rlwnm.**
rotlw	**rotlw** rA,rS,rB												equivalent to				**rlwnm** rA,rS,rB,**0,31**																	**rotlw**
rotlwi	**rotlwi** rA,rS,*n*												equivalent to				**rlwinm** rA,rS,*n*,**0,31**																	**rotlwi**
rotrwi	**rotrwi** rA,rS,*n*												equivalent to				**rlwinm** rA,rS,32 – *n*,**0,31**																	**rotrwi**
sc	0	1	0	0	0	1	///																								1	/	SC	**sc**
slw	0	1	1	1	1	1	rS						rA				rB				0	0	0	0	0	0	1	1	0	0	0	0	X	**slw**
slw.	0	1	1	1	1	1	rS						rA				rB				0	0	0	0	0	0	1	1	0	0	0	1	X	**slw.**
slwi	**slwi** rA,rS,*n* (n < 32)												equivalent to				**rlwinm** rA,rS,*n*,**0**,31 – *n*																	**slwi**
sraw	0	1	1	1	1	1	rS						rA				rB				1	1	0	0	0	0	1	1	0	0	0	0	X	**sraw**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
sraw.	0 1 1 1	1 1	rS	rA	rB	1 1 0	0 0 1 1	0 0 0 1	X	**sraw.**
srawi	0 1 1 1	1 1	rS	rA	SH	1 1 0	0 1 1 1	0 0 0 0	X	**srawi**
srawi.	0 1 1 1	1 1	rS	rA	SH	1 1 0	0 1 1 1	0 0 0 1	X	**srawi.**
srw	0 1 1 1	1 1	rS	rA	rB	1 0 0	0 0 1 1	0 0 0 0	X	**srw**
srw.	0 1 1 1	1 1	rS	rA	rB	1 0 0	0 0 1 1	0 0 0 1	X	**srw.**
srwi	**srwi rA,rS,*n*** (n < 32)		equivalent to		**rlwinm rA,rS,32 – *n*,*n*,31**					**srwi**
stb	1 0 0 1	1 0	rS	rA	D				D	**stb**
stbu	1 0 0 1	1 1	rS	rA	D				D	**stbu**
stbux	0 1 1 1	1 1	rS	rA	rB	0 0 1	1 1 1 0	1 1 1 0	X	**stbux**
stbx	0 1 1 1	1 1	rS	rA	rB	0 0 1	1 0 1 0	1 1 1 0	X	**stbx**
stfd	1 1 0 1	1 0	frS	rA	D				D	**stfd**
stfdu	1 1 0 1	1 1	frS	rA	D				D	**stfdu**
stfdux	0 1 1 1	1 1	frS	rA	rB	1 0 1	1 1 1 0	1 1 1 /	X	**stfdux**
stfdx	0 1 1 1	1 1	frS	rA	rB	1 0 1	1 0 1 0	1 1 1 /	X	**stfdx**
stfiwx [6]	0 1 1 1	1 1	frS	rA	rB	1 1 1	1 0 1 0	1 1 1 /	X	**stfiwx**
stfs	1 1 0 1	0 0	frS	rA	D				D	**stfs**
stfsu	1 1 0 1	0 1	frS	rA	D				D	**stfsu**
stfsux	0 1 1 1	1 1	frS	rA	rB	1 0 1	0 1 1 0	1 1 1 /	X	**stfsux**
stfsx	0 1 1 1	1 1	frS	rA	rB	1 0 1	0 0 1 0	1 1 1 /	X	**stfsx**
sth	1 0 1 1	0 0	rS	rA	D				D	**sth**
sthbrx	0 1 1 1	1 1	rS	rA	rB	1 1 1	0 0 1 0	1 1 0 /	X	**sthbrx**
sthu	1 0 1 1	0 1	rS	rA	D				D	**sthu**
sthux	0 1 1 1	1 1	rS	rA	rB	0 1 1	0 1 1 0	1 1 1 /	X	**sthux**
sthx	0 1 1 1	1 1	rS	rA	rB	0 1 1	0 0 1 0	1 1 1 /	X	**sthx**
stmw	1 0 1 1	1 1	rS	rA	D				D	**stmw**
stswi	0 1 1 1	1 1	rS	rA	NB	1 0 1	1 0 1 0	1 0 1 /	X	**stswi**
stswx	0 1 1 1	1 1	rS	rA	rB	1 0 1	0 0 1 0	1 0 1 /	X	**stswx**
stw	1 0 0 1	0 0	rS	rA	D				D	**stw**
stwbrx	0 1 1 1	1 1	rS	rA	rB	1 0 1	0 0 1 0	1 1 0 /	X	**stwbrx**
stwcx.	0 1 1 1	1 1	rS	rA	rB	0 0 1	0 0 1 0	1 1 0 1	X	**stwcx.**
stwu	1 0 0 1	0 1	rS	rA	D				D	**stwu**
stwux	0 1 1 1	1 1	rS	rA	rB	0 0 1	0 1 1 0	1 1 1 /	D	**stwux**
stwx	0 1 1 1	1 1	rS	rA	rB	0 0 1	0 0 1 0	1 1 1 /	D	**stwx**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
sub	**sub rD,rA,rB**		equivalent to	**subf rD,rB,rA**						**sub**
subc	**subc rD,rA,rB**		equivalent to	**subfc rD,rB,rA**						**subc**
subf	0 1 1 1	1 1	rD	rA	rB	0 0 0	0 1 0 1	0 0 0 0	X	**subf**
subf.	0 1 1 1	1 1	rD	rA	rB	0 0 0	0 1 0 1	0 0 0 1	X	**subf.**
subfc	0 1 1 1	1 1	rD	rA	rB	0 0 0	0 0 0 1	0 0 0 0	X	**subfc**
subfc.	0 1 1 1	1 1	rD	rA	rB	0 0 0	0 0 0 1	0 0 0 1	X	**subfc.**
subfco	0 1 1 1	1 1	rD	rA	rB	1 0 0	0 0 0 1	0 0 0 0	X	**subfco**
subfco.	0 1 1 1	1 1	rD	rA	rB	1 0 0	0 0 0 1	0 0 0 1	X	**subfco.**
subfe	0 1 1 1	1 1	rD	rA	rB	0 0 1	0 0 0 1	0 0 0 0	X	**subfe**
subfe.	0 1 1 1	1 1	rD	rA	rB	0 0 1	0 0 0 1	0 0 0 1	X	**subfe.**
subfeo	0 1 1 1	1 1	rD	rA	rB	1 0 1	0 0 0 1	0 0 0 0	X	**subfeo**
subfeo.	0 1 1 1	1 1	rD	rA	rB	1 0 1	0 0 0 1	0 0 0 1	X	**subfeo.**
subfic	0 0 1 0	0 0	rD	rA	SIMM				D	**subfic**
subfme	0 1 1 1	1 1	rD	rA	///	0 0 1	1 1 0 1	0 0 0 0	X	**subfme**
subfme.	0 1 1 1	1 1	rD	rA	///	0 0 1	1 1 0 1	0 0 0 1	X	**subfme.**
subfmeo	0 1 1 1	1 1	rD	rA	///	1 0 1	1 1 0 1	0 0 0 0	X	**subfmeo**
subfmeo.	0 1 1 1	1 1	rD	rA	///	1 0 1	1 1 0 1	0 0 0 1	X	**subfmeo.**
subfo	0 1 1 1	1 1	rD	rA	rB	1 0 0	0 1 0 1	0 0 0 0	X	**subfo**
subfo.	0 1 1 1	1 1	rD	rA	rB	1 0 0	0 1 0 1	0 0 0 1	X	**subfo.**
subfze	0 1 1 1	1 1	rD	rA	///	0 0 1	1 0 0 1	0 0 0 0	X	**subfze**
subfze.	0 1 1 1	1 1	rD	rA	///	0 0 1	1 0 0 1	0 0 0 1	X	**subfze.**
subfzeo	0 1 1 1	1 1	rD	rA	///	1 0 1	1 0 0 1	0 0 0 0	X	**subfzeo**
subfzeo.	0 1 1 1	1 1	rD	rA	///	1 0 1	1 0 0 1	0 0 0 1	X	**subfzeo.**
subi	**subi rD,rA,value**		equivalent to	**addi rD,rA,**–value						**subi**
subic	**subic rD,rA,value**		equivalent to	**addic rD,rA,**–value						**subic**
subic.	**subic. rD,rA,value**		equivalent to	**addic. rD,rA,**–value						**subic.**
subis	**subis rD,rA,value**		equivalent to	**addis rD,rA,**–value						**subis**
sync	0 1 1 1	1 1	///	///	///	1 0 0	1 0 1 0	1 1 0 0	X	**sync**
tlbia [6,7]	0 1 1 1	1 1	///	///	///	0 1 0	1 1 1 0	0 1 0 0	X	**tlbia**
tlbie [6,7]	0 1 1 1	1 1	///	///	rB	0 1 0	0 1 1 0	0 1 0 0	X	**tlbie**
tlbsync [6,7]	0 1 1 1	1 1	///	///	///	1 0 0	0 1 1 0	1 1 0 /	X	**tlbsync**
tw	0 1 1 1	1 1	TO	rA	rB	0 0 0	0 0 0 0	1 0 0 /	X	**tw**
tweq	**tweq rA,SIMM**		equivalent to	**tw 4,rA,**SIMM						**tweq**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31	Form	Mnemonic
tweqi	**tweqi rA,SIMM**		equivalent to	**twi 4,rA,SIMM**						**tweqi**
twge	**twge rA,SIMM**		equivalent to	**tw 12,rA,SIMM**						**twge**
twgei	**twgei rA,SIMM**		equivalent to	**twi 12,rA,SIMM**						**twgei**
twgt	**twgt rA,SIMM**		equivalent to	**tw 8,rA,SIMM**						**twgt**
twgti	**twgti rA,SIMM**		equivalent to	**twi 8,rA,SIMM**						**twgti**
twi	0 0 0 0	1 1	TO	rA	SIMM				D	**twi**
twle	**twle rA,SIMM**		equivalent to	**tw 20,rA,SIMM**						**twle**
twlei	**twlei rA,SIMM**		equivalent to	**twi 20,rA,SIMM**						**twlei**
twlge	**twlge rA,SIMM**		equivalent to	**tw 12,rA,SIMM**						**twlge**
twlgei	**twlgei rA,SIMM**		equivalent to	**twi 12,rA,SIMM**						**twlgei**
twlgt	**twlgt rA,SIMM**		equivalent to	**tw 1,rA,SIMM**						**twlgt**
twlgti	**twlgti rA,SIMM**		equivalent to	**twi 1,rA,SIMM**						**twlgti**
twlle	**twlle rA,SIMM**		equivalent to	**tw 6,rA,SIMM**						**twlle**
twllei	**twllei rA,SIMM**		equivalent to	**twi 6,rA,SIMM**						**twllei**
twllt	**twllt rA,SIMM**		equivalent to	**tw 2,rA,SIMM**						**twllt**
twllti	**twllti rA,SIMM**		equivalent to	**twi 2,rA,SIMM**						**twllti**
twlng	**twlng rA,SIMM**		equivalent to	**tw 6,rA,SIMM**						**twlng**
twlngi	**twlngi rA,SIMM**		equivalent to	**twi 6,rA,SIMM**						**twlngi**
twlnl	**twlnl rA,SIMM**		equivalent to	**tw 5,rA,SIMM**						**twlnl**
twlnli	**twlnli rA,SIMM**		equivalent to	**twi 5,rA,SIMM**						**twlnli**
twlt	**twlt rA,SIMM**		equivalent to	**tw 16,rA,SIMM**						**twlt**
twlti	**twlti rA,SIMM**		equivalent to	**twi 16,rA,SIMM**						**twlti**
twne	**twne rA,SIMM**		equivalent to	**tw 24,rA,SIMM**						**twne**
twnei	**twnei rA,SIMM**		equivalent to	**twi 24,rA,SIMM**						**twnei**
twng	**twng rA,SIMM**		equivalent to	**tw 20,rA,SIMM**						**twng**
twngi	**twngi rA,SIMM**		equivalent to	**twi 20,rA,SIMM**						**twngi**
twnl	**twnl rA,SIMM**		equivalent to	**tw 12,rA,SIMM**						**twnl**
twnli	**twnli rA,SIMM**		equivalent to	**twi 12,rA,SIMM**						**twnli**

Table A-3. Instructions Sorted by Mnemonic (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
xor	0	1	1	1	1	1	rS						rA				rB				0	1	0		0	1	1	1	1	0	0	0	X	**xor**
xor.	0	1	1	1	1	1	rS						rA				rB				0	1	0		0	1	1	1	1	0	0	1	X	**xor.**
xori	0	1	1	0	1	0	rS						rA				UIMM																D	**xori**
xoris	0	1	1	0	1	1	rS						rA				UIMM																D	**xoris**

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.
2 The value in the BI operand selects CR*n*[2], the EQ bit.
3 The value in the BI operand selects CR*n*[0], the LT bit.
4 The value in the BI operand selects CR*n*[1], the GT bit.
5 The value in the BI operand selects CR*n*[3], the SO bit.
6 Optional to the PowerPC classic architecture.
7 Supervisor-level instruction.
8 Access level is detemined by whether the SPR is defined as a user or supervisor level SPR.

A.4 Instructions Sorted by Opcode (Binary)

Table A-4 lists instructions by opcode, shown in binary.

Table A-4. Instructions Sorted by Opcode (Binary)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
twi	0	0	0	0	1	1	TO						rA				SIMM																D	**twi**
mulli	0	0	0	1	1	1	rD						rA				SIMM																D	**mulli**
subfic	0	0	1	0	0	0	rD						rA				SIMM																D	**subfic**
cmpli	0	0	1	0	1	0	crfD			/	L		rA				UIMM																D	**cmpli**
cmpi	0	0	1	0	1	1	crfD			/	L		rA				SIMM																D	**cmpi**
addic	0	0	1	1	0	0	rD						rA				SIMM																D	**addic**
addic.	0	0	1	1	0	1	rD						rA				SIMM																D	**addic.**
addi	0	0	1	1	1	0	rD						rA				SIMM																D	**addi**
addis	0	0	1	1	1	1	rD						rA				SIMM																D	**addis**
bc	0	1	0	0	0	0	BO						BI				BD														0	0	B	**bc**
bca	0	1	0	0	0	0	BO						BI				BD														1	0	B	**bca**
bcl	0	1	0	0	0	0	BO						BI				BD														0	1	B	**bcl**
bcla	0	1	0	0	0	0	BO						BI				BD														1	1	B	**bcla**
sc	0	1	0	0	0	1	///																								1	/	SC	**sc**
b	0	1	0	0	1	0	LI																								0	0	I	**b**
ba	0	1	0	0	1	0	LI																								1	0	I	**ba**
bl	0	1	0	0	1	0	LI																								0	1	I	**bl**

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
bla	0	1	0	0	1	0	LI																								1	1	I	**bla**
mcrf	0	1	0	0	1	1	crfD			//			crfS				///				0	0	0	0	0	0	0	0	0	0	0	/	XL	**mcrf**
bclr	0	1	0	0	1	1	BO						BI				///				0	0	0	0	0	1	0	0	0	0	0	0	XL	**bclr**
bclrl	0	1	0	0	1	1	BO						BI				///				0	0	0	0	0	1	0	0	0	0	0	1	XL	**bclrl**
crnor	0	1	0	0	1	1	crbD						crbA				crbB				0	0	0	0	0	1	0	0	0	0	1	/	XL	**crnor**
rfi [1]	0	1	0	0	1	1	///														0	0	0	0	0	1	1	0	0	1	0	/	XL	**rfi**
crandc	0	1	0	0	1	1	crbD						crbA				crbB				0	0	1	0	0	0	0	0	0	0	1	/	XL	**crandc**
isync	0	1	0	0	1	1	///														0	0	1	0	0	0	1	0	1	1	0	/	XL	**isync**
crxor	0	1	0	0	1	1	crbD						crbA				crbB				0	0	1	1	0	0	0	0	0	0	1	/	XL	**crxor**
crand	0	1	0	0	1	1	crbD						crbA				crbB				0	1	0	0	0	0	0	0	0	0	1	/	XL	**crand**
crnand	0	1	0	0	1	1	crbD						crbA				crbB				0	0	1	1	1	0	0	0	0	0	1	/	XL	**crnand**
creqv	0	1	0	0	1	1	crbD						crbA				crbB				0	1	0	0	1	0	0	0	0	0	1	/	XL	**creqv**
crorc	0	1	0	0	1	1	crbD						crbA				crbB				0	1	1	0	1	0	0	0	0	0	1	/	XL	**crorc**
cror	0	1	0	0	1	1	crbD						crbA				crbB				0	1	1	1	0	0	0	0	0	0	1	/	XL	**cror**
bcctr	0	1	0	0	1	1	BO						BI				///				1	0	0	0	0	1	0	0	0	0	0	0	XL	**bcctr**
bcctrl	0	1	0	0	1	1	BO						BI				///				1	0	0	0	0	1	0	0	0	0	0	1	XL	**bcctrl**
rlwimi	0	1	0	1	0	0	rS						rA				SH				MB				ME							0	M	**rlwimi**
rlwimi.	0	1	0	1	0	0	rS						rA				SH				MB				ME							1	M	**rlwimi.**
rlwinm	0	1	0	1	0	1	rS						rA				SH				MB				ME							0	M	**rlwinm**
rlwinm.	0	1	0	1	0	1	rS						rA				SH				MB				ME							1	M	**rlwinm.**
rlwnm	0	1	0	1	1	1	rS						rA				rB				MB				ME							0	M	**rlwnm**
rlwnm.	0	1	0	1	1	1	rS						rA				rB				MB				ME							1	M	**rlwnm.**
ori	0	1	1	0	0	0	rS						rA				UIMM																D	**ori**
oris	0	1	1	0	0	1	rS						rA				UIMM																D	**oris**
xori	0	1	1	0	1	0	rS						rA				UIMM																D	**xori**
xoris	0	1	1	0	1	1	rS						rA				UIMM																D	**xoris**
andi.	0	1	1	1	0	0	rS						rA				UIMM																D	**andi.**
andis.	0	1	1	1	0	1	rS						rA				UIMM																D	**andis.**
cmp	0	1	1	1	1	1	crfD			/	L		rA				rB				0	0	0	0	0	0	0	0	0	0	0	/	X	**cmp**
tw	0	1	1	1	1	1	TO						rA				rB				0	0	0	0	0	0	0	0	1	0	0	/	X	**tw**
subfc	0	1	1	1	1	1	rD						rA				rB				0	0	0	0	0	0	0	1	0	0	0	0	X	**subfc**
subfc.	0	1	1	1	1	1	rD						rA				rB				0	0	0	0	0	0	0	1	0	0	0	1	X	**subfc.**
addc	0	1	1	1	1	1	rD						rA				rB				0	0	0	0	0	0	0	1	0	1	0	0	X	**addc**

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
addc.	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	0	0	1	0	1	0	1	X	addc.
mulhwu	0	1	1	1	1	1	rD						rA				rB				/	0	0		0	0	0	1	0	1	1	0	X	mulhwu
mulhwu.	0	1	1	1	1	1	rD						rA				rB				/	0	0		0	0	0	1	0	1	1	1	X	mulhwu.
mfcr	0	1	1	1	1	1	rD						///								0	0	0		0	0	1	0	0	1	1	/	X	mfcr
lwarx	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	0	1	0	1	0	0	/	X	lwarx
lwzx	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	0	1	0	1	1	1	/	X	lwzx
slw	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	0	1	1	0	0	0	0	X	slw
slw.	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	0	1	1	0	0	0	1	X	slw.
cntlzw	0	1	1	1	1	1	rS						rA				///				0	0	0		0	0	1	1	0	1	0	0	X	cntlzw
cntlzw.	0	1	1	1	1	1	rS						rA				///				0	0	0		0	0	1	1	0	1	0	1	X	cntlzw.
and	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	0	1	1	1	0	0	0	X	and
and.	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	0	1	1	1	0	0	1	X	and.
cmpl	0	1	1	1	1	1	/	L	rA				rB				///				0	0	0		0	1	0	0	0	0	0	/	X	cmpl
subf	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	1	0	1	0	0	0	0	X	subf
subf.	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	1	0	1	0	0	0	1	X	subf.
dcbst	0	1	1	1	1	1	///						rA				rB				0	0	0		0	1	1	0	1	1	0	/	X	dcbst
lwzux	0	1	1	1	1	1	rD						rA				rB				0	0	0		0	1	1	0	1	1	1	/	X	lwzux
andc	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	1	1	1	1	0	0	0	X	andc
andc.	0	1	1	1	1	1	rS						rA				rB				0	0	0		0	1	1	1	1	0	0	1	X	andc.
mulhw	0	1	1	1	1	1	rD						rA				rB				/	0	0		1	0	0	1	0	1	1	0	X	mulhw
mulhw.	0	1	1	1	1	1	rD						rA				rB				/	0	0		1	0	0	1	0	1	1	1	X	mulhw.
mfmsr [1]	0	1	1	1	1	1	rD						///								0	0	0		1	0	1	0	0	1	1	/	X	mfmsr
dcbf	0	1	1	1	1	1	///						rA				rB				0	0	0		1	0	1	0	1	1	0	/	X	dcbf
lbzx	0	1	1	1	1	1	rD						rA				rB				0	0	0		1	0	1	0	1	1	1	/	X	lbzx
neg	0	1	1	1	1	1	rD						rA				///				0	0	0		1	1	0	1	0	0	0	0	X	neg
neg.	0	1	1	1	1	1	rD						rA				///				0	0	0		1	1	0	1	0	0	0	1	X	neg.
lbzux	0	1	1	1	1	1	rD						rA				rB				0	0	0		1	1	1	0	1	1	1	/	X	lbzux
nor	0	1	1	1	1	1	rS						rA				rB				0	0	0		1	1	1	1	1	0	0	0	X	nor
nor.	0	1	1	1	1	1	rS						rA				rB				0	0	0		1	1	1	1	1	0	0	1	X	nor.
subfe	0	1	1	1	1	1	rD						rA				rB				0	0	1		0	0	0	1	0	0	0	0	X	subfe
subfe.	0	1	1	1	1	1	rD						rA				rB				0	0	1		0	0	0	1	0	0	0	1	X	subfe.
adde	0	1	1	1	1	1	rD						rA				rB				0	0	1		0	0	0	1	0	1	0	0	X	adde
adde.	0	1	1	1	1	1	rD						rA				rB				0	0	1		0	0	0	1	0	1	0	1	X	adde.

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mtcrf	0	1	1	1	1	1	rS					/	CRM								/	0	0	1	0	0	1	0	0	0	0	/	XFX	**mtcrf**
mtmsr [1]	0	1	1	1	1	1	rS					///										0	0	1	0	0	1	0	0	1	0	/	X	**mtmsr**
stwcx.	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	0	1	X	**stwcx.**
stwx	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	0	1	0	1	1	1	/	D	**stwx**
stwux	0	1	1	1	1	1	rS					rA					rB					0	0	1	0	1	1	0	1	1	1	/	D	**stwux**
subfze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	0	X	**subfze**
subfze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	0	0	1	X	**subfze.**
addze	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	0	X	**addze**
addze.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	0	0	1	0	1	0	1	X	**addze.**
mtsr [1]	0	1	1	1	1	1	rS					/	SR				///					0	0	1	1	0	1	0	0	1	0	0	X	**mtsr**
stbx	0	1	1	1	1	1	rS					rA					rB					0	0	1	1	0	1	0	1	1	1	0	X	**stbx**
subfme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	0	X	**subfme**
subfme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	0	0	1	X	**subfme.**
addme	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	0	X	**addme**
addme.	0	1	1	1	1	1	rD					rA					///					0	0	1	1	1	0	1	0	1	0	1	X	**addme.**
mullw	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	0	X	**mullw**
mullw.	0	1	1	1	1	1	rD					rA					rB					0	0	1	1	1	0	1	0	1	1	1	X	**mullw.**
mtsrin [1]	0	1	1	1	1	1	rS					///					rB					0	0	1	1	1	1	0	0	1	0	0	X	**mtsrin**
dcbtst	0	1	1	1	1	1	CT					rA					rB					0	0	1	1	1	1	0	1	1	0	/	X	**dcbtst**
stbux	0	1	1	1	1	1	rS					rA					rB					0	0	1	1	1	1	0	1	1	1	0	X	**stbux**
add	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	0	X	**add**
add.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	0	1	0	1	0	1	X	**add.**
dcbt	0	1	1	1	1	1	CT					rA					rB					0	1	0	0	0	1	0	1	1	0	/	X	**dcbt**
lhzx	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	0	1	1	1	/	X	**lhzx**
eqv	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	0	X	**eqv**
eqv.	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	0	1	1	1	0	0	1	X	**eqv.**
tlbie [1,2]	0	1	1	1	1	1	///					///					rB					0	1	0	0	1	1	0	0	1	0	0	X	**tlbie**
eciwx [2]	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	1	1	0	1	1	0	0	X	**eciwx**
lhzux	0	1	1	1	1	1	rD					rA					rB					0	1	0	0	1	1	0	1	1	1	/	X	**lhzux**
xor	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	0	X	**xor**
xor.	0	1	1	1	1	1	rS					rA					rB					0	1	0	0	1	1	1	1	0	0	1	X	**xor.**
mfspr [3]	0	1	1	1	1	1	rD					SPR[5–9]					SPR[0–4]					0	1	0	1	0	1	0	0	1	1	/	XFX	**mfspr**
lhax	0	1	1	1	1	1	rD					rA					rB					0	1	0	1	0	1	0	1	1	1	/	X	**lhax**

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
tlbia [1, 2]	0	1	1	1	1	1	///						///				///				0	1	0		1	1	1	0	0	1	0	0	X	**tlbia**
mftb	0	1	1	1	1	1	rD						TBR[5–9]				TBR[0–4]				0	1	0		1	1	1	0	0	1	1	0	XFX	**mftb**
lhaux	0	1	1	1	1	1	rD						rA				rB				0	1	0		1	1	1	0	1	1	1	/	X	**lhaux**
sthx	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	0	1	0	1	1	1	/	X	**sthx**
orc	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	0	1	1	1	0	0	0	X	**orc**
orc.	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	0	1	1	1	0	0	1	X	**orc.**
ecowx [2]	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	1	1	0	1	1	0	0	X	**ecowx**
sthux	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	1	1	0	1	1	1	/	X	**sthux**
or	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	1	1	1	1	0	0	0	X	**or**
or.	0	1	1	1	1	1	rS						rA				rB				0	1	1		0	1	1	1	1	0	0	1	X	**or.**
divwu	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	0	0	1	0	1	1	0	X	**divwu**
divwu.	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	0	0	1	0	1	1	1	X	**divwu.**
mtspr [3]	0	1	1	1	1	1	rS						SPR[5–9]				SPR[0–4]				0	1	1		1	0	1	0	0	1	1	/	XFX	**mtspr**
dcbi [1]	0	1	1	1	1	1	///						rA				rB				0	1	1		1	0	1	0	1	1	0	/	X	**dcbi**
nand	0	1	1	1	1	1	rS						rA				rB				0	1	1		1	0	1	1	1	0	0	0	X	**nand**
nand.	0	1	1	1	1	1	rS						rA				rB				0	1	1		1	0	1	1	1	0	0	1	X	**nand.**
divw	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	1	0	1	0	1	1	0	X	**divw**
divw.	0	1	1	1	1	1	rD						rA				rB				0	1	1		1	1	0	1	0	1	1	1	X	**divw.**
mcrxr	0	1	1	1	1	1	crfD						///								1	0	0		0	0	0	0	0	0	0	/	X	**mcrxr**
subfco	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	0	1	0	0	0	0	X	**subfco**
subfco.	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	0	1	0	0	0	1	X	**subfco.**
addco	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	0	1	0	1	0	0	X	**addco**
addco.	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	0	1	0	1	0	1	X	**addco.**
lswx	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	1	0	1	0	1	/	X	**lswx**
lwbrx	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	0	1	0	1	1	0	/	X	**lwbrx**
lfsx	0	1	1	1	1	1	frD						rA				rB				1	0	0		0	0	1	0	1	1	1	/	X	**lfsx**
srw	0	1	1	1	1	1	rS						rA				rB				1	0	0		0	0	1	1	0	0	0	0	X	**srw**
srw.	0	1	1	1	1	1	rS						rA				rB				1	0	0		0	0	1	1	0	0	0	1	X	**srw.**
subfo	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	1	0	1	0	0	0	0	X	**subfo**
subfo.	0	1	1	1	1	1	rD						rA				rB				1	0	0		0	1	0	1	0	0	0	1	X	**subfo.**
tlbsync [1,2]	0	1	1	1	1	1	///						///				///				1	0	0		0	1	1	0	1	1	0	/	X	**tlbsync**
lfsux	0	1	1	1	1	1	frD						rA				rB				1	0	0		0	1	1	0	1	1	1	/	X	**lfsux**
mfsr [1]	0	1	1	1	1	1	rD					/	SR				///				1	0	0		1	0	1	0	0	1	1	0	X	**mfsr**

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
lswi	0	1	1	1	1	1	rD						rA				NB				1	0	0		1	0	1	0	1	0	1	/	X	**lswi**
sync	0	1	1	1	1	1	///						///				///				1	0	0		1	0	1	0	1	1	0	0	X	**sync**
lfdx	0	1	1	1	1	1	frD						rA				rB				1	0	0		1	0	1	0	1	1	1	/	X	**lfdx**
nego	0	1	1	1	1	1	rD						rA				///				1	0	0		1	1	0	1	0	0	0	0	X	**nego**
nego.	0	1	1	1	1	1	rD						rA				///				1	0	0		1	1	0	1	0	0	0	1	X	**nego.**
lfdux	0	1	1	1	1	1	frD						rA				rB				1	0	0		1	1	1	0	1	1	1	/	X	**lfdux**
subfeo	0	1	1	1	1	1	rD						rA				rB				1	0	1		0	0	0	1	0	0	0	0	X	**subfeo**
subfeo.	0	1	1	1	1	1	rD						rA				rB				1	0	1		0	0	0	1	0	0	0	1	X	**subfeo.**
addeo	0	1	1	1	1	1	rD						rA				rB				1	0	1		0	0	0	1	0	1	0	0	X	**addeo**
addeo.	0	1	1	1	1	1	rD						rA				rB				1	0	1		0	0	0	1	0	1	0	1	X	**addeo.**
mfsrin [1]	0	1	1	1	1	1	rD						///				rB				1	0	1		0	0	1	0	0	1	1	0	X	**mfsrin**
stswx	0	1	1	1	1	1	rS						rA				rB				1	0	1		0	0	1	0	1	0	1	/	X	**stswx**
stwbrx	0	1	1	1	1	1	rS						rA				rB				1	0	1		0	0	1	0	1	1	0	/	X	**stwbrx**
stfsx	0	1	1	1	1	1	frS						rA				rB				1	0	1		0	0	1	0	1	1	1	/	X	**stfsx**
stfsux	0	1	1	1	1	1	frS						rA				rB				1	0	1		0	1	1	0	1	1	1	/	X	**stfsux**
subfzeo	0	1	1	1	1	1	rD						rA				///				1	0	1		1	0	0	1	0	0	0	0	X	**subfzeo**
subfzeo.	0	1	1	1	1	1	rD						rA				///				1	0	1		1	0	0	1	0	0	0	1	X	**subfzeo.**
addzeo	0	1	1	1	1	1	rD						rA				///				1	0	1		1	0	0	1	0	1	0	0	X	**addzeo**
addzeo.	0	1	1	1	1	1	rD						rA				///				1	0	1		1	0	0	1	0	1	0	1	X	**addzeo.**
stswi	0	1	1	1	1	1	rS						rA				NB				1	0	1		1	0	1	0	1	0	1	/	X	**stswi**
stfdx	0	1	1	1	1	1	frS						rA				rB				1	0	1		1	0	1	0	1	1	1	/	X	**stfdx**
subfmeo	0	1	1	1	1	1	rD						rA				///				1	0	1		1	1	0	1	0	0	0	0	X	**subfmeo**
subfmeo.	0	1	1	1	1	1	rD						rA				///				1	0	1		1	1	0	1	0	0	0	1	X	**subfmeo.**
addmeo	0	1	1	1	1	1	rD						rA				///				1	0	1		1	1	0	1	0	1	0	0	X	**addmeo**
addmeo.	0	1	1	1	1	1	rD						rA				///				1	0	1		1	1	0	1	0	1	0	1	X	**addmeo.**
mullwo	0	1	1	1	1	1	rD						rA				rB				1	0	1		1	1	0	1	0	1	1	0	X	**mullwo**
mullwo.	0	1	1	1	1	1	rD						rA				rB				1	0	1		1	1	0	1	0	1	1	1	X	**mullwo.**
dcba [2]	0	1	1	1	1	1	///						rA				rB				1	0	1		1	1	1	0	1	1	0	/	X	**dcba**
stfdux	0	1	1	1	1	1	frS						rA				rB				1	0	1		1	1	1	0	1	1	1	/	X	**stfdux**
addo	0	1	1	1	1	1	rD						rA				rB				1	1	0		0	0	0	1	0	1	0	0	X	**addo**
addo.	0	1	1	1	1	1	rD						rA				rB				1	1	0		0	0	0	1	0	1	0	1	X	**addo.**
lhbrx	0	1	1	1	1	1	rD						rA				rB				1	1	0		0	0	1	0	1	1	0	/	X	**lhbrx**
sraw	0	1	1	1	1	1	rS						rA				rB				1	1	0		0	0	1	1	0	0	0	0	X	**sraw**

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
sraw.	0	1	1	1	1	1	rS						rA				rB				1	1	0		0	0	1	1	0	0	0	1	X	sraw.
srawi	0	1	1	1	1	1	rS						rA				SH				1	1	0		0	1	1	1	0	0	0	0	X	srawi
srawi.	0	1	1	1	1	1	rS						rA				SH				1	1	0		0	1	1	1	0	0	0	1	X	srawi.
eieio	0	1	1	1	1	1	///						///				///				1	1	0		1	0	1	0	1	1	0	0	X	eieio
sthbrx	0	1	1	1	1	1	rS						rA				rB				1	1	1		0	0	1	0	1	1	0	/	X	sthbrx
extsh	0	1	1	1	1	1	rS						rA				///				1	1	1		0	0	1	1	0	1	0	0	X	extsh
extsh.	0	1	1	1	1	1	rS						rA				///				1	1	1		0	0	1	1	0	1	0	1	X	extsh.
extsb	0	1	1	1	1	1	rS						rA				///				1	1	1		0	1	1	1	0	1	0	0	X	extsb
extsb.	0	1	1	1	1	1	rS						rA				///				1	1	1		0	1	1	1	0	1	0	1	X	extsb.
divwuo	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	0	0	1	0	1	1	0	X	divwuo
divwuo.	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	0	0	1	0	1	1	1	X	divwuo.
icbi	0	1	1	1	1	1	///						rA				rB				1	1	1		1	0	1	0	1	1	0	/	X	icbi
stfiwx [2]	0	1	1	1	1	1	frS						rA				rB				1	1	1		1	0	1	0	1	1	1	/	X	stfiwx
divwo	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	1	0	1	0	1	1	0	X	divwo
divwo.	0	1	1	1	1	1	rD						rA				rB				1	1	1		1	1	0	1	0	1	1	1	X	divwo.
dcbz	0	1	1	1	1	1	///						rA				rB				1	1	1		1	1	1	0	1	1	0	/	X	dcbz
lwz	1	0	0	0	0	0	rD						rA				D																D	lwz
lwzu	1	0	0	0	0	1	rD						rA				D																D	lwzu
lbz	1	0	0	0	1	0	rD						rA				D																D	lbz
lbzu	1	0	0	0	1	1	rD						rA				D																D	lbzu
stw	1	0	0	1	0	0	rS						rA				D																D	stw
stwu	1	0	0	1	0	1	rS						rA				D																D	stwu
stb	1	0	0	1	1	0	rS						rA				D																D	stb
stbu	1	0	0	1	1	1	rS						rA				D																D	stbu
lhz	1	0	1	0	0	0	rD						rA				D																D	lhz
lhzu	1	0	1	0	0	1	rD						rA				D																D	lhzu
lha	1	0	1	0	1	0	rD						rA				D																D	lha
lhau	1	0	1	0	1	1	rD						rA				D																D	lhau
sth	1	0	1	1	0	0	rS						rA				D																D	sth
sthu	1	0	1	1	0	1	rS						rA				D																D	sthu
lmw	1	0	1	1	1	0	rD						rA				D																D	lmw
stmw	1	0	1	1	1	1	rS						rA				D																D	stmw
lfs	1	1	0	0	0	0	frD						rA				D																D	lfs

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
lfsu	1	1	0	0	0	1	frD						rA				D																D	**lfsu**
lfd	1	1	0	0	1	0	frD						rA				D																D	**lfd**
lfdu	1	1	0	0	1	1	frD						rA				D																D	**lfdu**
stfs	1	1	0	1	0	0	frS						rA				D																D	**stfs**
stfsu	1	1	0	1	0	1	frS						rA				D																D	**stfsu**
stfd	1	1	0	1	1	0	frS						rA				D																D	**stfd**
stfdu	1	1	0	1	1	1	frS						rA				D																D	**stfdu**
fdivs	1	1	1	0	1	1	frD						frA				frB				///				1	0			0	1	0	0	A	**fdivs**
fdivs.	1	1	1	0	1	1	frD						frA				frB				///				1	0			0	1	0	1	A	**fdivs.**
fsubs	1	1	1	0	1	1	frD						frA				frB				///				1	0			1	0	0	0	A	**fsubs**
fsubs.	1	1	1	0	1	1	frD						frA				frB				///				1	0			1	0	0	1	A	**fsubs.**
fadds	1	1	1	0	1	1	frD						frA				frB				///				1	0			1	0	1	0	A	**fadds**
fadds.	1	1	1	0	1	1	frD						frA				frB				///				1	0			1	0	1	1	A	**fadds.**
fsqrts [2]	1	1	1	0	1	1	frD						///				frB				///				1	0			1	1	0	0	A	**fsqrts**
fsqrts. [2]	1	1	1	0	1	1	frD						///				frB				///				1	0			1	1	0	1	A	**fsqrts.**
fres [2]	1	1	1	0	1	1	frD						///				frB				///				1	1			0	0	0	0	A	**fres**
fres. [2]	1	1	1	0	1	1	frD						///				frB				///				1	1			0	0	0	1	A	**fres.**
fmuls	1	1	1	0	1	1	frD						frA				///				frC				1	1			0	0	1	0	A	**fmuls**
fmuls.	1	1	1	0	1	1	frD						frA				///				frC				1	1			0	0	1	1	A	**fmuls.**
fmsubs	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	0	0	0	A	**fmsubs**
fmsubs.	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	0	0	1	A	**fmsubs.**
fmadds	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	0	1	0	A	**fmadds**
fmadds.	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	0	1	1	A	**fmadds.**
fnmsubs	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	1	0	0	A	**fnmsubs**
fnmsubs.	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	1	0	1	A	**fnmsubs.**
fnmadds	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	1	1	0	A	**fnmadds**
fnmadds.	1	1	1	0	1	1	frD						frA				frB				frC				1	1			1	1	1	1	A	**fnmadds.**
fcmpu	1	1	1	1	1	1	crfD			//			frA				frB				0	0	0	0	0	0	0	0	0	0	0	/	X	**fcmpu**
frsp	1	1	1	1	1	1	frD						///				frB				0	0	0	0	0	0	0	1	1	0	0	0	X	**frsp**
frsp.	1	1	1	1	1	1	frD						///				frB				0	0	0	0	0	0	0	1	1	0	0	1	X	**frsp.**
fctiw	1	1	1	1	1	1	frD						///				frB				0	0	0	0	0	0	0	1	1	1	0	0	X	**fctiw**
fctiw.	1	1	1	1	1	1	frD						///				frB				0	0	0	0	0	0	0	1	1	1	0	1	X	**fctiw.**
fctiwz	1	1	1	1	1	1	frD						///				frB				0	0	0	0	0	0	0	1	1	1	1	0	X	**fctiwz**

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
fctiwz.	1	1	1	1	1	1	frD						///				frB				0	0	0		0	0	0	1	1	1	1	1	X	**fctiwz.**
fdiv	1	1	1	1	1	1	frD						frA				frB				///						1	0	0	1	0	0	A	**fdiv**
fdiv.	1	1	1	1	1	1	frD						frA				frB				///						1	0	0	1	0	1	A	**fdiv.**
fsub	1	1	1	1	1	1	frD						frA				frB				///						1	0	1	0	0	0	A	**fsub**
fsub.	1	1	1	1	1	1	frD						frA				frB				///						1	0	1	0	0	1	A	**fsub.**
fadd	1	1	1	1	1	1	frD						frA				frB				///						1	0	1	0	1	0	A	**fadd**
fadd.	1	1	1	1	1	1	frD						frA				frB				///						1	0	1	0	1	1	A	**fadd.**
fsqrt [2]	1	1	1	1	1	1	frD						///				frB				///						1	0	1	1	0	0	A	**fsqrt**
fsqrt. [2]	1	1	1	1	1	1	frD						///				frB				///						1	0	1	1	0	1	A	**fsqrt.**
fsel [2]	1	1	1	1	1	1	frD						frA				frB				frC						1	0	1	1	1	0	A	**fsel**
fsel. [2]	1	1	1	1	1	1	frD						frA				frB				frC						1	0	1	1	1	1	A	**fsel.**
fmul	1	1	1	1	1	1	frD						frA				///				frC						1	1	0	0	1	0	A	**fmul**
fmul.	1	1	1	1	1	1	frD						frA				///				frC						1	1	0	0	1	1	A	**fmul.**
frsqrte [2]	1	1	1	1	1	1	frD						///				frB				///						1	1	0	1	0	0	A	**frsqrte**
frsqrte. [2]	1	1	1	1	1	1	frD						///				frB				///						1	1	0	1	0	1	A	**frsqrte.**
fmsub	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	0	0	A	**fmsub**
fmsub.	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	0	1	A	**fmsub.**
fmadd	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	1	0	A	**fmadd**
fmadd.	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	0	1	1	A	**fmadd.**
fnmsub	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	1	0	0	A	**fnmsub**
fnmsub.	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	1	0	1	A	**fnmsub.**
fnmadd	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	1	1	0	A	**fnmadd**
fnmadd.	1	1	1	1	1	1	frD						frA				frB				frC						1	1	1	1	1	1	A	**fnmadd.**
fcmpo	1	1	1	1	1	1	crfD		//				frA				frB				0	0	0		0	1	0	0	0	0	0	/	X	**fcmpo**
mtfsb1	1	1	1	1	1	1	crbD						///								0	0	0		0	1	0	0	1	1	0	0	X	**mtfsb1**
mtfsb1.	1	1	1	1	1	1	crbD						///								0	0	0		0	1	0	0	1	1	0	1	X	**mtfsb1.**
fneg	1	1	1	1	1	1	frD						///				frB				0	0	0		0	1	0	1	0	0	0	0	X	**fneg**
fneg.	1	1	1	1	1	1	frD						///				frB				0	0	0		0	1	0	1	0	0	0	1	X	**fneg.**
mcrfs	1	1	1	1	1	1	crfD		//				crfS				///				0	0	0		1	0	0	0	0	0	0	/	X	**mcrfs**
mtfsb0	1	1	1	1	1	1	crbD						///								0	0	0		1	0	0	0	1	1	0	0	X	**mtfsb0**
mtfsb0.	1	1	1	1	1	1	crbD						///								0	0	0		1	0	0	0	1	1	0	1	X	**mtfsb0.**
fmr	1	1	1	1	1	1	frD						///				frB				0	0	0		1	0	0	1	0	0	0	0	A	**fmr**
fmr.	1	1	1	1	1	1	frD						///				frB				0	0	0		1	0	0	1	0	0	0	1	A	**fmr.**

Table A-4. Instructions Sorted by Opcode (Binary) (continued)

Mnemonic	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	Form	Mnemonic
mtfsfi	1	1	1	1	1	1	crfD				///						IMM				/	0	0	1	0	0	0	0	1	1	0	0	X	**mtfsfi**
mtfsfi.	1	1	1	1	1	1	crfD				///						IMM				/	0	0	1	0	0	0	0	1	1	0	1	X	**mtfsfi.**
fnabs	1	1	1	1	1	1	frD						///				frB					0	0	1	0	0	0	1	0	0	0	0	X	**fnabs**
fnabs.	1	1	1	1	1	1	frD						///				frB					0	0	1	0	0	0	1	0	0	0	1	X	**fnabs.**
fabs	1	1	1	1	1	1	frD						///				frB					0	1	0	0	0	0	1	0	0	0	0	X	**fabs**
fabs.	1	1	1	1	1	1	frD						///				frB					0	1	0	0	0	0	1	0	0	0	1	X	**fabs.**
mffs	1	1	1	1	1	1	frD						///									1	0	0	1	0	0	0	1	1	1	0	X	**mffs**
mffs.	1	1	1	1	1	1	frD						///									1	0	0	1	0	0	0	1	1	1	1	X	**mffs.**
mtfsf	1	1	1	1	1	1	/	FM								/	frB					1	0	1	1	0	0	0	1	1	1	0	XFL	**mtfsf**
mtfsf.	1	1	1	1	1	1	/	FM								/	frB					1	0	1	1	0	0	0	1	1	1	1	XFL	**mtfsf.**

1 Supervisor-level instruction

2 Optional to the PowerPC classic architecture

3 Access level is detemined by whether the SPR is defined as a user or supervisor level SPR.

A.5 Instruction Set Legend

Table A-5 provides general information on the instruction set (such as architectural level, privilege level, and form).

Table A-5. PowerPC Instruction Set Legend

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
add*x*	√					XO	**add***x*
addc*x*	√					XO	**addc***x*
adde*x*	√					XO	**adde***x*
addi	√					D	**addi**
addic	√					D	**addic**
addic.	√					D	**addic.**
addis	√					D	**addis**
addme*x*	√					XO	**addme***x*
addze*x*	√					XO	**addze***x*
and*x*	√					X	**and***x*
andc*x*	√					X	**andc***x*
andi.	√					D	**andi.**
andis.	√					D	**andis.**
b*x*	√					I	**b***x*

Table A-5. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
bc*x*	√					B	**bc***x*
bcctr*x*	√					XL	**bcctr***x*
bclr*x*	√					XL	**bclr***x*
cmp	√					X	**cmp**
cmpi	√					D	**cmpi**
cmpl	√					X	**cmpl**
cmpli	√					D	**cmpli**
cntlzw*x*	√					X	**cntlzw***x*
crand	√					XL	**crand**
crandc	√					XL	**crandc**
creqv	√					XL	**creqv**
crnand	√					XL	**crnand**
crnor	√					XL	**crnor**
cror	√					XL	**cror**
crorc	√					XL	**crorc**
crxor	√					XL	**crxor**
dcba		√			√	X	**dcba**
dcbf		√				X	**dcbf**
dcbi			√	√		X	**dcbi**
dcbst		√				X	**dcbst**
dcbt		√				X	**dcbt**
dcbtst		√				X	**dcbts**t
dcbz		√				X	**dcbz**
divw*x*	√					XO	**divw***x*
divwu*x*	√					XO	**divwu***x*
eciwx		√			√	X	**eciwx**
ecowx		√			√	X	**ecowx**
eieio		√				X	**eieio**
eqv*x*	√					X	**eqv***x*
extsb*x*	√					X	**extsb***x*
extsh*x*	√					X	**extsh***x*
fabs*x*	√					X	**fabs***x*
fadd*x*	√					A	**fadd***x*

Table A-5. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
fadds*x*	√					A	**fadds***x*
fcmpo	√					X	**fcmpo**
fcmpu	√					X	**fcmpu**
fctiw*x*	√					X	**fctiw***x*
fctiwz*x*	√					X	**fctiwz***x*
fdiv*x*	√					A	**fdiv***x*
fdivs*x*	√					A	**fdivs***x*
fmadd*x*	√					A	**fmadd***x*
fmadds*x*	√					A	**fmadds***x*
fmr*x*	√					X	**fmr***x*
fmsub*x*	√					A	**fmsub***x*
fmsubs*x*	√					A	**fmsubs***x*
fmul*x*	√					A	**fmul***x*
fmuls*x*	√					A	**fmuls***x*
fnabs*x*	√					X	**fnabs***x*
fneg*x*	√					X	**fneg***x*
fnmadd*x*	√					A	**fnmadd***x*
fnmadds*x*	√					A	**fnmadds***x*
fnmsub*x*	√					A	**fnmsub***x*
fnmsubs*x*	√					A	**fnmsubs***x*
fres*x*	√				√	A	**fres***x*
frsp*x*	√					X	**frsp***x*
frsqrte*x*	√				√	A	**frsqrte***x*
fsel*x*	√				√	A	**fsel***x*
fsqrt*x*	√				√	A	**fsqrt***x*
fsqrts*x*	√				√	A	**fsqrts***x*
fsub*x*	√					A	**fsub***x*
fsubs*x*	√					A	**fsubs***x*
icbi		√				X	**icbi**
isync		√				XL	**isync**
lbz	√					D	**lbz**
lbzu	√					D	**lbzu**
lbzux	√					X	**lbzux**

Table A-5. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
lbzx	√					X	**lbzx**
lfd	√					D	**lfd**
lfdu	√					D	**lfdu**
lfdux	√					X	**lfdux**
lfdx	√					X	**lfdx**
lfs	√					D	**lfs**
lfsu	√					D	**lfsu**
lfsux	√					X	**lfsux**
lfsx	√					X	**lfsx**
lha	√					D	**lha**
lhau	√					D	**lhau**
lhaux	√					X	**lhaux**
lhax	√					X	**lhax**
lhbrx	√					X	**lhbrx**
lhz	√					D	**lhz**
lhzu	√					D	**lhzu**
lhzux	√					X	**lhzux**
lhzx	√					X	**lhzx**
lmw [1]	√					D	**lmw** [2]
lswi [1]	√					X	**lswi** [1]
lswx[1]	√					X	**lswx**[1]
lwarx	√					X	**lwarx**
lwbrx	√					X	**lwbrx**
lwz	√					D	**lwz**
lwzu	√					D	**lwzu**
lwzux	√					X	**lwzux**
lwzx	√					X	**lwzx**
mcrf	√					XL	**mcrf**
mcrfs	√					X	**mcrfs**
mcrxr	√					X	**mcrx**r
mfcr	√					X	**mfcr**
mffs	√					X	**mffs**
mfmsr			√	√		X	**mfmsr**

Table A-5. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
mfspr[3]	√		√	√		XFX	**mfspr**[3]
mfsr			√	√		X	**mfsr**
mfsrin			√	√		X	**mfsrin**
mftb		√				XFX	**mftb**
mtcrf	√					XFX	**mtcrf**
mtfsb0*x*	√					X	**mtfsb0***x*
mtfsb1*x*	√					X	**mtfsb1***x*
mtfsf*x*	√					XFL	**mtfsf***x*
mtfsfi*x*	√					X	**mtfsfi***x*
mtmsr			√	√		X	**mtmsr**
mtspr[3]	√		√	√		XFX	**mtspr**[4]
mtsr			√	√		X	**mtsr**
mtsrin			√	√		X	**mtsrin**
mulhw*x*	√					XO	**mulhw***x*
mulhwu*x*	√					XO	**mulhwu***x*
mulli	√					D	**mulli**
mullw*x*	√					XO	**mullw***x*
nand*x*	√					X	**nand***x*
neg*x*	√					XO	**neg***x*
nor*x*	√					X	**nor***x*
or*x*	√					X	**or***x*
orc*x*	√					X	**orc***x*
ori	√					D	**ori**
oris	√					D	**oris**
rfi			√	√		XL	**rfi**
rlwimi*x*	√					M	**rlwimi***x*
rlwinm*x*	√					M	**rlwinm***x*
rlwnm*x*	√					M	**rlwnm***x*
sc	√		√			SC	**sc**
slw*x*	√					X	**slw***x*
sraw*x*	√					X	**sraw***x*
srawi*x*	√					X	**srawi***x*
srw*x*	√					X	**srw***x*

Table A-5. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
stb	√					D	**stb**
stbu	√					D	**stbu**
stbux	√					X	**stbux**
stbx	√					X	**stbx**
stfd	√					D	**stfd**
stfdu	√					D	**stfdu**
stfdux	√					X	**stfdux**
stfdx	√					X	**stfdx**
stfiwx	√					X	**stfiwx**
stfs	√					D	**stfs**
stfsu	√					D	**stfsu**
stfsux	√					X	**stfsux**
stfsx	√					X	**stfsx**
sth	√					D	**sth**
sthbrx	√					X	**sthbrx**
sthu	√					D	**sthu**
sthux	√					X	**sthux**
sthx	√					X	**sthx**
stmw [1]	√					D	**stmw** [1]
stswi [1]	√					X	**stswi** [1]
stswx [1]	√					X	**stswx** [1]
stw	√					D	**stw**
stwbrx	√					X	**stwbrx**
stwcx.	√					X	**stwcx.**
stwu	√					D	**stwu**
stwux	√					X	**stwux**
stwx	√					X	**stwx**
subf*x*	√					XO	**subf***x*
subfc*x*	√					XO	**subfc***x*
subfe*x*	√					XO	**subfe***x*
subfic	√					D	**subfic**
subfme*x*	√					XO	**subfme***x*
subfze*x*	√					XO	**subfze***x*

Table A-5. PowerPC Instruction Set Legend (continued)

	UISA	VEA	OEA	Supervisor Level	Optional	Form	
sync	√					X	**sync**
tlbia*x*			√	√	√	X	**tlbia***x*
tlbie*x*			√	√	√	X	**tlbie***x*
tlbsync			√	√	√	X	**tlbsync**
tw	√					X	**tw**
twi	√					D	**tw**i
xor*x*	√					X	**xor***x*
xori	√					D	**xori**
xoris	√					D	**xoris**

1 Load/Store string or multiple.
2 Load/Store string or multiple.
3 Supervisor- and user-level instruction
4 Supervisor- and user-level instruction

Appendix B
Multiple-Precision Shifts

This appendix gives examples of how multiple precision shifts can be programmed.

B.1 Overview

A multiple-precision shift is initially defined to be a shift of an *n*-word quantity, where $n > 1$. The quantity to be shifted is contained in *n* registers. The shift amount is specified either by an immediate value in the instruction or by bits 27-31 of a register.

The examples shown below distinguish between the cases $n = 2$ and $n > 2$. However if $n > 2$, the shift amount must be in the range 0–31 for the examples to yield the desired result. The specific instance shown for $n > 2$ is $n = 3$: extending those instruction sequences to larger *n* is straightforward, as is reducing them to the case $n = 2$ when the more stringent restriction on shift amount is met. For shifts with immediate shift amounts, only the case $n = 3$ is shown because the more stringent restriction on shift amount is always met.

In the examples it is assumed that GPRs 2 and 3 (and 4) contain the quantity to be shifted, and that the result is to be placed into the same registers. For non-immediate shifts, the shift amount is assumed to be in bits 27-31 of GPR6. For immediate shifts, the shift amount is assumed to be greater than zero. GPRs 0–31 are used as scratch registers. For $n > 2$, the number of instructions required is $2n - 1$ (immediate shifts) or $3n - 1$ (non-immediate shifts).

The following sections provide examples of multiple-precision shifts.

B.2 Multiple-Precision Shifts in 32-Bit Implementations

Shift Left Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm   r2,r2,sh,0,31 – sh
rlwimi   r2,r3,sh,32 – sh,31
rlwinm   r3,r3,sh,0,31 – sh
rlwimi   r3,r4,sh,32 – sh,31
rlwinm   r4,r4,sh,0,31 – sh
```

Shift Left, *n* = 2 (Shift Amount < 64)

```
subfic   r31,r6,32
slw      r2,r2,r6
srw      r0,r3,r31
or       r2,r2,r0
addi     r31,r6,–32
slw      r0,r3,r31
or       r2,r2,r0
slw      r3,r3,r6
```

Shift Left, *n* = 3 (Shift Amount < 32)

```
subfic   r31,r6,32
slw      r2,r2,r6
srw      r0,r3,r31
or       r2,r2,r0
slw      r3,r3,r6
srw      r0,r4,r31
or       r3,r3,r0
slw      r4,r4,r6
```

Shift Right Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm   r4,r4,32 – sh,sh,31
rlwimi   r4,r3,32 – sh,0,sh – 1
rlwinm   r3,r3,32 – sh,sh,31
rlwimi   r3,r2,32 – sh,0,sh – 1
rlwinm   r2,r2,32 – sh,sh,31
```

Shift Right, *n* = 2 (Shift Amount < 64)

```
subfic   r31,r6,32
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
addi     r31,r6, –32
srw      r0,r2,r31
or       r3,r3,r0
srw      r2,r2,r6
```

Shift Right, *n* = 3 (Shift Amount < 32)

```
subfic   r31,r6,–32
srw      r4,r4,r6
slw      r0,r3,r31
or       r4,r4,r0
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
srw      r2,r2,r6
```

Shift Right Algebraic Immediate, *n* = 3 (Shift Amount < 32)

```
rlwinm   r4,r4,32 – sh,sh,31
rlwimi   r4,r3,32 – sh,0,sh – 1
rlwinm   r3,r3,32 – sh,sh,31
rlwimi   r3,r2,32 – sh,0,sh – 1
srawi    r2,r2,sh
```

Shift Right Algebraic, *n* = 2 (Shift Amount < 64)

```
subfic   r31,r6,32
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
addic.   r31,r6,–32
sraw     r0,r2,r31
ble      $+8
ori      r3,r0,0
sraw     r2,r2,r6
```

Shift Right Algebraic, *n* = 3 (Shift Amount < 32)

```
subfic   r31,r6,32
srw      r4,r4,r6
slw      r0,r3,r31
or       r4,r4,r0
srw      r3,r3,r6
slw      r0,r2,r31
or       r3,r3,r0
sraw     r2,r2,r6
```

Appendix C
Floating-Point Models

This appendix describes the execution model for IEEE operations and gives examples of how the floating-point conversion instructions can be used to perform various conversions as well as providing models for floating-point instructions.

C.1 Execution Model for IEEE Operations

The following description uses double-precision arithmetic as an example; single-precision arithmetic is similar except that the fraction field is a 23-bit field and the single-precision guard, round, and sticky bits (described in this section) are logically adjacent to the 23-bit FRACTION field. IEEE-conforming significand arithmetic is performed with a floating-point accumulator where bits 0–55, shown in Figure C-1, comprise the significand of the intermediate result.

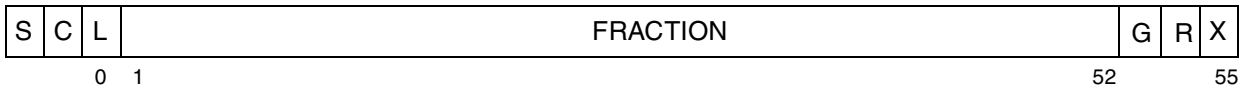


Figure C-1. IEEE 64-Bit Execution Model

The bits and fields for the IEEE double-precision execution model are defined in Table C-1.

Table C-1. IEEE 64-Bit Execution Model Field Descriptions

Bits	Description
S	Sign bit.
C	Carry bit that captures the carry out of the significand.
L	Leading unit bit of the significand that receives the implicit bit from the operands.
FRACTION	52-bit field that accepts the fraction of the operands.
G R X	Guard, round, and sticky. These bits are extensions to the low-order accumulator bits. G and R are required for postnormalization of the result. G, R, and X are required during rounding to determine if the intermediate result is equally near the two nearest representable values. This is shown in Table C-2. X serves as an extension to the G and R bits by representing the logical OR of all bits that may appear to the low-order side of the R bit, due to either shifting the accumulator right or to other generation of low-order result bits. G and R participate in the left shifts with zeros being shifted into the R bit.

Table C-2 shows the significance of G, R, and bits with respect to the intermediate result (IR), the next lower in magnitude representable number (NL), and the next higher in magnitude representable number (NH).

Table C-2. Interpretation of G, R, and X Bits

G	R	X	Interpretation
0	0	0	IR is exact
0	0	1	
0	1	0	IR closer to NL
0	1	1	
1	0	0	IR midway between NL & NH
1	0	1	
1	1	0	IR closer to NH
1	1	1	

The significand of the intermediate result is made up of the L bit, the FRACTION, and the G, R, and X bits.

The infinitely precise intermediate result of an operation is the result normalized in bits L, FRACTION, G, R, and X of the floating-point accumulator.

After normalization, the intermediate result is rounded, using the rounding mode specified by FPSCR[RN]. If rounding causes a carry into C, the significand is shifted right one position and the exponent is incremented by one. This causes an inexact result and possibly exponent overflow. Fraction bits to the left of the bit position used for rounding are stored into the FPR, and low-order bit positions, if any, are set to zero.

Four user-selectable rounding modes are provided through FPSCR[RN] as described in Section 3.3.5, “Rounding.” For rounding, the conceptual guard, round, and sticky bits are defined in terms of accumulator bits.

Table C-3 shows the positions of the guard, round, and sticky bits for double-precision and single-precision floating-point numbers in the IEEE execution model.

Table C-3. Location of the Guard, Round, and Sticky Bits—IEEE Execution Model

Format	Guard	Round	Sticky
Double	G bit	R bit	X bit
Single	24	25	OR of 26–52 G,R,X

Rounding can be treated as though the significand were shifted right, if required, until the least-significant bit to be retained is in the low-order bit position of the FRACTION. If any of the guard, round, or sticky bits are nonzero, the result is inexact.

Z1 and Z2, defined in Section 3.3.5, “Rounding,” can be used to approximate the result in the target format when one of the following rules is used:

- Round to nearest
  - Guard bit = 0: The result is truncated. (Result exact (GRX = 000) or closest to next lower value in magnitude (GRX = 001, 010, or 011).

— Guard bit = 1: Depends on round and sticky bits:

Case a: If the round or sticky bit is one (inclusive), the result is incremented (result closest to next higher value in magnitude (GRX = 101, 110, or 111)).

Case b: If the round and sticky bits are zero (result midway between closest representable values) then if the low-order bit of the result is one, the result is incremented. Otherwise (the low-order bit of the result is zero) the result is truncated (this is the case of a tie rounded to even).

If during the round-to-nearest process, truncation of the unrounded number produces the maximum magnitude for the specified precision, the following occurs:

— Guard bit = 1: Store infinity with the sign of the unrounded result.

— Guard bit = 0: Store the truncated (maximum magnitude) value.

- Round toward zero—Choose the smaller in magnitude of Z1 or Z2. If the guard, round, or sticky bit is nonzero, the result is inexact.
- Round toward +infinity—Choose Z1.
- Round toward –infinity—Choose Z2.

Where a result is to have fewer than 53 bits of precision because the instruction is a floating round to single-precision or single-precision arithmetic instruction, the intermediate result either is normalized or is placed in correct denormalized form before being rounded.

C.2 Multiply-Add Type Instruction Execution Model

The architecture uses of a special instruction form that performs up to three operations in one instruction (a multiply, an add, and a negate). With this comes an ability to produce a more exact intermediate result as an input to the rounder. Single-precision arithmetic is similar except that the fraction field is smaller. Note that rounding occurs only after add; therefore, the computation of the sum and product together are infinitely precise before the final result is rounded to a representable format.

As Figure C-2 shows, multiply-add significand arithmetic is considered to be performed with a floating-point accumulator, where bits 1–106 comprise the significand of the intermediate result.

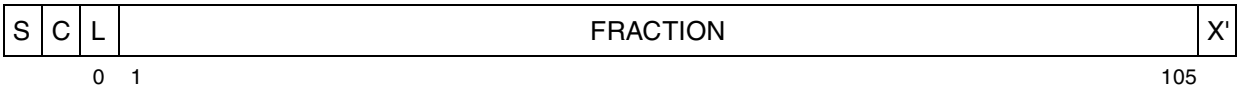


Figure C-2. Multiply-Add 64-Bit Execution Model

The first part of the operation is a multiply. The multiply has two 53-bit significands as inputs, which are assumed to be prenormalized, and produces a result conforming to the above model. If there is a carry out of the significand (into the C bit), the significand is shifted right one position, placing the L bit into the most-significant bit of the FRACTION and placing the C bit into the L bit. All 106 bits (L bit plus the fraction) of the product take part in the add operation. If the exponents of the two inputs to the adder are not equal, the significand of the operand with the smaller exponent is aligned (shifted) to the right by an amount added to that exponent to make it equal to the other input's exponent. Zeros are shifted into the left of the significand as it is aligned and bits shifted out of bit 105 of the significand are ORed into the X' bit. The add operation also produces a result conforming to the above model with the X' bit taking part in the add operation.

The result of the add is then normalized, with all bits of the add result, except the X' bit, participating in the shift. The normalized result serves as the intermediate result that is input to the rounder.

For rounding, the conceptual guard, round, and sticky bits are defined in terms of accumulator bits. Figure C-4 shows the positions of these bits for double-precision and single-precision floating-point numbers in the multiply-add execution model.

Table C-4. Location of G, R, and X Bits—Multiply-Add Execution Model

Format	Guard	Round	Sticky
Double	53	54	OR of 55–105, X'
Single	24	25	OR of 26–105, X'

The rules for rounding the intermediate result are the same as those given in Section C.1, “Execution Model for IEEE Operations.”

If the instruction is floating negative multiply-add or floating negative multiply-subtract, the final result is negated.

Floating-point multiply-add instructions combine a multiply and an add operation without an intermediate rounding operation. The fraction part of the intermediate product is 106 bits wide, and all 106 bits take part in the add/subtract portion of the instruction.

Status bits are set as follows:

- Overflow, underflow, and inexact exception bits, the FR and FI bits, and the FPRF field are set based on the final result of the operation, not on the result of the multiplication.
- Invalid operation exception bits are set as if the multiplication and the addition were performed using two separate instructions (for example, an **fmul** instruction followed by an **fadd** instruction). That is, multiplication of infinity by 0 or of anything by an SNaN, causes the corresponding exception bits to be set.

C.3 Floating-Point Conversions

This section provides examples of floating-point conversion instructions. Note that some of the examples use the optional Floating Select (**fsel)** instruction. Care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities.

C.3.1 Conversion from Floating-Point Number to Signed Fixed-Point Integer Word

The full convert to signed fixed-point integer word function can be implemented with the following sequence, assuming that the floating-point value to be converted is in FPR1, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fctiw[z] f2,f1             #convert to fx int
stfd     f2,disp(r1)       #store float
lwa      r3,disp + 4(r1)   #load word algebraic
                           #(use lwz on a 32-bit implementation)
```

C.3.2 Conversion from Floating-Point Number to Unsigned Fixed-Point Integer Word

In a 32-bit implementation, the full convert to unsigned fixed-point integer word function can be implemented with the sequence below, assuming that the floating-point value to be converted is in FPR1, the value zero is in FPR0, the value $2^{32} - 1$ is in FPR3, the value 2^{31} is in FPR4, the result is returned in GPR3, and a double word at displacement (disp) from the address in GPR1 can be used as scratch space.

```
fsel     f2,f1,f1,f0       #use 0 if < 0
fsub     f5,f3,f1          #use max if > max
fsel     f2,f5,f2,f3
fsub     f5,f2,f4          #subtract 2**31
fcmpu    cr2,f2,f4         #use diff if ≥ 2**31
fsel     f2,f5,f5,f2
fctiw[z] f2,f2             #convert to fx int
stfd     f2,disp(r1)       #store float
lwz      r3,disp + 4(r1)   #load word
blt      cr2,$+8           #add 2**31 if input
xoris    r3,r3,0x8000      #was ≥ 2**31
```

C.4 Floating-Point Models

This section describes models for floating-point instructions.

C.4.1 Floating-Point Round to Single-Precision Model

The following describes the operation of the **frsp** instruction.

```
If frB[1–11] < 897 and frB[1–63] > 0 then
    Do
      If FPSCR[UE] = 0 then goto Disabled Exponent Underflow
      If FPSCR[UE] = 1 then goto Enabled Exponent Underflow
    End

If frB[1–11] > 1150 and frB[1–11] < 2047 then
    Do
      If FPSCR[OE] = 0 then goto Disabled Exponent Overflow
      If FPSCR[OE] = 1 then goto Enabled Exponent Overflow
    End

If frB[1–11] > 896 and frB[1–11] < 1151 then goto Normal Operand

If frB[1–63] = 0 then goto Zero Operand

If frB[1–11] = 2047 then
    Do
      If frB[12–63] = 0 then goto Infinity Operand
      If frB[12] = 1 then goto QNaN Operand
      If frB[12] = 0 and frB[13–63] > 0 then goto SNaN Operand
    End
```

Disabled Exponent Underflow:

```
sign ← frB[0]
If frB[1–11] = 0 then
    Do
      exp ← –1022
      frac[0–52] ← 0b0 || frB[12–63]
    End
```

```
If frB[1–11] > 0 then
      Do
        exp ← frB[1–11] – 1023
        frac[0–52] ← 0b1 || frB[12–63]
      End
Denormalize operand:
      G || R || X ← 0b000
      Do while exp < –126
        exp ← exp + 1
        frac[0–52] || G || R || X ← 0b0 || frac || G || (R | X)
      End
FPSCR[UX] ← frac[24–52] || G || R || X > 0
Round single(sign,exp,frac[0–52],G,R,X)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
If frac[0–52] = 0 then
      Do
        frD[0] ← sign
        frD[1–63] ← 0
        If sign = 0 then FPSCR[FPRF] ← “+zero”
        If sign = 1 then FPSCR[FPRF] ← “–zero”
      End
If frac[0–52] > 0 then
      Do
        If frac[0] = 1 then
          Do
            If sign = 0 then FPSCR[FPRF] ← “+normal number”
            If sign = 1 then FPSCR[FPRF] ← “–normal number”
          End
        If frac[0] = 0 then
          Do
            If sign = 0 then FPSCR[FPRF] ← “+denormalized number”
            If sign = 1 then FPSCR[FPRF] ← “–denormalized number”
          End
        Normalize operand:
          Do while frac[0] = 0
            exp ← exp – 1
            frac[0–52] ← frac[1–52] || 0b0
          End
        frD[0] ← sign
        frD[1–11] ← exp + 1023
        frD[12–63] ← frac[1–52]
      End
Done
```

Enabled Exponent Underflow

```
FPSCR[UX] ← 1
sign ← frB[0]
If frB[1–11] = 0 then
      Do
        exp ← –1022
        frac[0–52] ← 0b0 || frB[12–63]
      End
If frB[1–11] > 0 then
      Do
        exp ← frB[1–11] – 1023
        frac[0–52] ← 0b1 || frB[12–63]
      End

Normalize operand:
      Do while frac[0] = 0
        exp ← exp – 1
        frac[0–52] ← frac[1–52] || 0b0
      End
```

```
Round single(sign,exp,frac[0–52],0,0,0)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
exp ← exp + 192
frD[0] ← sign
frD[1–11] ← exp + 1023
frD[12–63] ← frac[1–52]
If sign = 0 then FPSCR[FPRF] ← “+normal number”
If sign = 1 then FPSCR[FPRF] ← “–normal number”
Done
```

Disabled Exponent Overflow

```
FPSCR[OX] ← 1
If FPSCR[RN] = 0b00 then          /* Round to Nearest */
    Do
      If frB[0] = 0 then frD ← 0x7FF0_0000_0000_0000
      If frB[0] = 1 then frD ← 0xFFF0_0000_0000_0000
      If frB[0] = 0 then FPSCR[FPRF] ← “+infinity”
      If frB[0] = 1 then FPSCR[FPRF] ← “–infinity”
    End
If FPSCR[RN] = 0b01 then          /* Round Truncate */
    Do
      If frB[0] = 0 then frD ← 0x47EF_FFFF_E000_0000
      If frB[0] = 1 then frD ← 0xC7EF_FFFF_E000_0000
      If frB[0] = 0 then FPSCR[FPRF] ← “+normal number”
      If frB[0] = 1 then FPSCR[FPRF] ← “–normal number”
    End
If FPSCR[RN] = 0b10 then          /* Round to +Infinity */
    Do
      If frB[0] = 0 then frD ← 0x7FF0_0000_0000_0000
      If frB[0] = 1 then frD ← 0xC7EF_FFFF_E000_0000
      If frB[0] = 0 then FPSCR[FPRF] ← “+infinity”
      If frB[0] = 1 then FPSCR[FPRF] ← “–normal number”
    End
If FPSCR[RN] = 0b11 then          /* Round to -Infinity */
    Do
      If frB[0] = 0 then frD ← 0x47EF_FFFF_E000_0000
      If frB[0] = 1 then frD ← 0xFFF0_0000_0000_0000
      If frB[0] = 0 then FPSCR[FPRF] ← “+normal number”
      If frB[0] = 1 then FPSCR[FPRF] ← “–infinity”
    End
FPSCR[FR] ← undefined
FPSCR[FI] ← 1
FPSCR[XX] ← 1
Done
```

Enabled Exponent Overflow

```
sign ← frB[0]
exp ← frB[1–11] – 1023
    frac[0–52] ← 0b1 || frB[12–63]
    Round single(sign,exp,frac[0–52],0,0,0)
    FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
Enabled Overflow
    FPSCR[OX] ← 1
    exp ← exp – 192
    frD[0] ← sign
    frD[1–11] ← exp + 1023
    frD[12–63] ← frac[1–52]
    If sign = 0 then FPSCR[FPRF] ← “+normal number”
    If sign = 1 then FPSCR[FPRF] ← “–normal number”
Done
```

Zero Operand

```
frD ← frB
If frB[0] = 0 then FPSCR[FPRF] ← “+zero”
If frB[0] = 1 then FPSCR[FPRF] ← “–zero”
FPSCR[FR FI] ← 0b00
Done
```

Infinity Operand

```
frD ← frB
If frB[0] = 0 then FPSCR[FPRF] ← “+infinity”
If frB[0] = 1 then FPSCR[FPRF] ← “–infinity”
Done
```

QNaN Operand

```
frD ← frB[0–34] || 0b0_0000_0000_0000_0000_0000_0000_0000
FPSCR[FPRF] ← “QNaN”
FPSCR[FR FI] ← 0b00
Done
```

SNaN Operand

```
FPSCR[VXSNAN] ← 1
If FPSCR[VE] = 0 then
    Do
      frD[0–11] ← frB[0–11]
      frD[12] ← 1
      frD[13–63] ← frB[13–34] || 0b0_0000_0000_0000_0000_0000_0000_0000
      FPSCR[FPRF] ← “QNaN”
    End
FPSCR[FR FI] ← 0b00
Done
```

Normal Operand

```
sign ← frB[0]
exp ← frB[1–11] – 1023
frac[0–52] ← 0b1 || frB[12–63]
Round single(sign,exp,frac[0–52],0,0,0)
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
If exp > +127 and FPSCR[OE] = 0 then go to Disabled Exponent Overflow
If exp > +127 and FPSCR[OE] = 1 then go to Enabled Overflow
frD[0] ← sign
frD[1–11] ← exp + 1023
frD[12–63] ← frac[1–52]
If sign = 0 then FPSCR[FPRF] ← “+normal number”
If sign = 1 then FPSCR[FPRF] ← “–normal number”
Done
```

Round Single (sign,exp,frac[0–52],G,R,X)

```
inc ← 0
lsb ← frac[23]
gbit ← frac[24]
rbit ← frac[25]
xbit ← (frac[26–52] || G || R || X) ≠ 0
If FPSCR[RN] = 0b00 then
    Do
      If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
      If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If FPSCR[RN] = 0b10 then
```

```
      Do
        If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
      End
If FPSCR[RN] = 0b11 then
      Do
        If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
        If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
      End
frac[0–23] ← frac[0–23] + inc
If carry_out =1 then
      Do
        frac[0–23] ← 0b1 || frac[0–22]
        exp ← exp + 1
      End
frac[24–52] ← (29)0
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

C.4.2 Floating-Point Convert to Integer Model

The following algorithm describes the operation of the floating-point convert to integer instructions. In this example, ‘u’ represents an undefined hexadecimal digit.

```
If Floating Convert to Integer Word
      Then Do
        Then round_mode ← FPSCR[RN]
        tgt_precision ← “32-bit integer”
      End
If Floating Convert to Integer Word with round toward Zero
      Then Do
        round_mode ← 0b01
        tgt_precision ← “32-bit integer”
      End
If Floating Convert to Integer Double Word
      Then Do
        round_mode ← FPSCR[RN]
        tgt_precision ← “64-bit integer”
      End
If Floating Convert to Integer Double Word with Round toward Zero
      Then Do
        round_mode ← 0b01
        tgt_precision ← “64-bit integer”
      End
sign ← frB[0]
If frB[1–11] = 2047 and frB[12–63] = 0 then goto Infinity Operand
If frB[1–11] = 2047 and frB[12] = 0 then goto SNaN Operand
If frB[1–11] = 2047 and frB[12] = 1 then goto QNaN Operand
If frB[1–11] > 1054 then goto Large Operand


If frB[1–11] > 0 then exp ← frB[1–11] – 1023 /* exp – bias */
If frB[1–11] = 0 then exp ← –1022
If frB[1–11] > 0 then frac[0–64]← 0b01 || frB[12–63] || (11)0 /*normal*/
If frB[1–11] = 0 then frac[0–64]← 0b00 || frB[12–63] || (11)0 /*denormal*/


gbit || rbit || xbit ← 0b000
Do i = 1,63 – exp                    /*do the loop 0 times if exp = 63*/
      frac[0–64] || gbit || rbit || xbit ← 0b0 || frac[0–64] || gbit || (rbit | xbit)
End
```

Round Integer (sign,frac[0–64],gbit,rbit,xbit,round_mode)

If sign = 1 then frac[0–64] ← ¬frac[0–64] + 1 /* needed leading 0 for $-2^{64} < frB < -2^{63}$*/

If tgt_precision = “32-bit integer” and frac[0–64] > $+2^{31} - 1$
    then goto Large Operand
If tgt_precision = “64-bit integer” and frac[0–64] > $+2^{63} - 1$
    then goto Large Operand
If tgt_precision = “32-bit integer” and frac[0–64] < -2^{31} then goto Large Operand

FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]

If tgt_precision = “64-bit integer” and frac[0–64] < -2^{63} then goto Large Operand
If tgt_precision = “32-bit integer”
    then frD ← 0xxuuu_uuuu || frac[33–64]
If tgt_precision = “64-bit integer” then frD ← frac[1–64]
FPSCR[FPRF] ← undefined
Done

Round Integer(sign,frac[0–64],gbit,rbit,xbit,round_mode)

```
/* comparisons ignore U bits*/
inc ← 0
If round_mode = 0b00 then
    Do
      If sign || frac[64] || gbit || rbit || xbit = 0bu11uu then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = 0bu011u then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = 0bu01u1 then inc ← 1
    End
If round_mode = 0b10 then
    Do
        If sign || frac[64] || gbit || rbit || xbit = 0b0u1uu then inc ←1
        If sign || frac[64] || gbit || rbit || xbit = 0b0uu1u then inc ← 1
        If sign || frac[64] || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
      End
If round_mode = 0b11 then
    Do
      If sign || frac[64] || gbit || rbit || xbit = 0b1u1uu then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = 0b1uu1u then inc ← 1
      If sign || frac[64] || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
    End
frac[0–64] ← frac[0–64] + inc
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
Return
```

Infinity Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then Do
    If tgt_precision = “32-bit integer” then
      Do
        If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
        If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
      End
    Else
      Do
        If sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
        If sign = 1 then frD ← 0x8000_0000_0000_0000
      End
    FPSCR[FPRF] ← undefined
    End
Done
```

SNaN Operand

```
FPSCR[FR FI VXCVI VXSNAN] ← 0b0011
If FPSCR[VE] = 0 then
      Do
        If tgt_precision = “32-bit integer”
          then frD ← 0xuuuu_uuuu_8000_0000
        If tgt_precision = “64-bit integer”
          then frD ← 0x8000_0000_0000_0000
        FPSCR[FPRF] ← undefined
      End
Done
```

QNaN Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then
      Do
        If tgt_precision = “32-bit integer” then frD ← 0xuuuu_uuuu_8000_0000
        If tgt_precision = “64-bit integer” then frD ← 0x8000_0000_0000_0000
        FPSCR[FPRF] ← undefined
      End
Done
```

Large Operand

```
FPSCR[FR FI VXCVI] ← 0b001
If FPSCR[VE] = 0 then Do
      If tgt_precision = “32-bit integer” then
        Do
          If sign = 0 then frD ← 0xuuuu_uuuu_7FFF_FFFF
          If sign = 1 then frD ← 0xuuuu_uuuu_8000_0000
        End
      Else
        Do
          If sign = 0 then frD ← 0x7FFF_FFFF_FFFF_FFFF
          If sign = 1 then frD ← 0x8000_0000_0000_0000
        End
      FPSCR[FPRF] ← undefined
      End
Done
```

C.4.3 Floating-Point Convert from Integer Model

The following describes the operation of floating-point convert from integer instructions.

```
sign ← frB[0]
exp ← 63
frac[0–63] ← frB

If frac[0–63] = 0 then go to Zero Operand

If sign = 1 then frac[0–63] ← ¬frac[0–63] + 1

Do while frac[0] = 0
      frac[0–63] ← frac[1–63] || '0'
      exp ← exp – 1
End

Round Float(sign,exp,frac[0–63],FPSCR[RN])

If sign = 1 then FPSCR[FPRF] ← “–normal number”
If sign = 0 then FPSCR[FPRF] ← “+normal number”
frD[0] ← sign
frD[1–11] ← exp + 1023
frD[12–63] ← frac[1–52]
Done
```

Zero Operand

```
FPSCR[FR FI] ← 0b00
FPSCR[FPRF] ← “+zero”
frD ← 0x0000_0000_0000_0000
Done
```

Round Float(sign,exp,frac[0–63],round_mode)

```
/* comparisons ignore U bits*/
inc ← 0
lsb ← frac[52]
gbit ← frac[53]
rbit ← frac[54]
xbit ← frac[55–63] > 0
If round_mode = 0b00 then
      Do

         If sign || lsb || gbit || rbit || xbit = 0bu11uu then inc ← 1
         If sign || lsb || gbit || rbit || xbit = 0bu011u then inc ← 1
         If sign || lsb || gbit || rbit || xbit = 0bu01u1 then inc ← 1
      End
If round_mode = 0b10 then
      Do
         If sign || lsb || gbit || rbit || xbit = 0b0u1uu then inc ← 1
         If sign || lsb || gbit || rbit || xbit = 0b0uu1u then inc ← 1
         If sign || lsb || gbit || rbit || xbit = 0b0uuu1 then inc ← 1
      End
If round_mode = 0b11 then
      Do
         If sign || lsb || gbit || rbit || xbit = 0b1u1uu then inc ← 1
         If sign || lsb || gbit || rbit || xbit = 0b1uu1u then inc ← 1
         If sign || lsb || gbit || rbit || xbit = 0b1uuu1 then inc ← 1
      End
frac[0–52] ← frac[0–52] + inc
If carry_out = 1 then exp ← exp + 1
FPSCR[FR] ← inc
FPSCR[FI] ← gbit | rbit | xbit
FPSCR[XX] ← FPSCR[XX] | FPSCR[FI]
Return
```

C.5 Floating-Point Selection

The following are examples of how the optional **fsel** instruction can be used to implement floating-point minimum and maximum functions, and certain simple forms of if-then-else constructions, without branching.

The examples show program fragments in an imaginary, C-like, high-level programming language, and the corresponding program fragment using **fsel** and other PowerPC instructions. In the examples, floating-point variables *a*, *b*, *x*, *y*, and *z* are assumed to be in FPRs *fa*, *fb*, *fx*, *fy*, and *fz*. FPR *fs* is assumed to be available for scratch space.

Additional examples can be found in Section C.3, “Floating-Point Conversions.”

Note that care must be taken in using **fsel** if IEEE compatibility is required, or if the values being tested can be NaNs or infinities; see Section C.5.4, “Notes.”

C.5.1 Comparison to Zero

This section provides examples in a program fragment code sequence for the comparison to zero case.

High-level language:	PowerPC:
`if a ≥ 0.0 then x ← y` `else x ← z`	`fsel fx,fa,fy,fz(see Section C.5.4, “Notes” number 1)`
`if a > 0.0 then x ← y` `else x ← z`	`fneg fs,fa` `fsel fx,fs,fz,fy(see Section C.5.4, “Notes” numbers 1 and 2)`

`if a = 0.0 then x ← y`
`else x ← z`
	`fsel fx,fa,fy,fz`
	`fneg fs,fa`
	`fsel fx,fs,fx,fz(see Section C.5.4, “Notes” number 1)`

C.5.2 Minimum and Maximum

This section provides program fragment code examples for minimum and maximum cases.

High-level language:	PowerPC:
`x ← min(a, b)`	`fsub fs,fa,fb(see Section C.5.4, “Notes” numbers 3, 4, and 5)` `fsel fx,fs,fb,fa`
`x ← max(a, b)`	`fsub fs,fa,fb(see Section C.5.4, “Notes” numbers 3, 4, and 5)` `fsel fx,fs,fa,fb`

C.5.3 Simple If-Then-Else Constructions

This section provides examples in a program fragment code sequence for simple if-then-else statements.

High-level language:	PowerPC:
`if a ≥ b then x ← y` `else x ← z`	`fsub fs,fa,fb` `fsel fx,fs,fy,fz(see Section C.5.4, “Notes” numbers 4 and 5)`
`if a > b then x ← y` `else x ← z`	`fsub fs,fb,fa` `fsel fx,fs,fz,fy(see Section C.5.4, “Notes” numbers 3, 4, and 5)`

`if a = b then x ← y`
`else x ← z`
	`fsub fs,fa,fb`
	`fsel fx,fs,fy,fz`
	`fneg fs,fs`
	`fsel fx,fs,fx,fz(see Section C.5.4, “Notes” numbers 4 and 5)`

C.5.4 Notes

The following notes apply to the examples found in Section C.5.1, “Comparison to Zero,” Section C.5.2, “Minimum and Maximum,” and Section C.5.3, “Simple If-Then-Else Constructions,” and to the corresponding cases using the other three arithmetic relations (<, ≤, and ≠). These notes should also be considered when any other use of **fsel** is contemplated.

In these notes the optimized program is the program shown and the unoptimized program (not shown) is the corresponding program that uses **fcmpu** and branch conditional instructions instead of **fsel**.

1. The unoptimized program affects FPSCR[VXSNAN] and so may cause the system error handler to be invoked if the corresponding exception is enabled; the optimized program does not affect this bit. This property of the optimized program is incompatible with the IEEE standard.
2. The optimized program gives the incorrect result if ‘a’ is a NaN.
3. The optimized program gives the incorrect result if ‘a’ and/or ‘b’ is a NaN (except that it may give the correct result in some cases for the minimum and maximum functions, depending on how those functions are defined to operate on NaNs).
4. The optimized program gives the incorrect result if ‘a’ and ‘b’ are infinities of the same sign. (Here it is assumed that invalid operation exceptions are disabled, in which case the result of the subtraction is a NaN. The analysis is more complicated if invalid operation exceptions are enabled, because in that case the target register of the subtraction is unchanged.)
5. The optimized program affects FPSCR[OX,UX,XX,VXISI], and therefore may cause the system error handler to be invoked if the corresponding exceptions are enabled, while the unoptimized program does not affect these bits. This property of the optimized program is incompatible with the IEEE standard.

C.6 Floating-Point Load Instructions

There are two basic forms of load instruction—single-precision and double-precision. Because FPRs support only double-precision format, single-precision load floating-point instructions convert single-precision data to double-precision format. The conversion and loading steps follow:

Let WORD[0–31] be the floating point single-precision operand accessed from memory.

Normalized Operand

```
If WORD[1–8] > 0 and WORD[1–8] < 255
  frD[0–1] ← WORD[0–1]
  frD[2] ← ¬ WORD[1]
  frD[3] ← ¬ WORD[1]
  frD[4] ← ¬ WORD[1]
  frD[5–63] ← WORD[2–31] || (29)0
```

Denormalized Operand

```
If WORD[1–8] = 0 and WORD[9–31] ≠ 0
  sign ← WORD[0]
  exp ← –126
  frac[0–52] ← 0b0 || WORD[9–31] || (29)0
  normalize the operand
    Do while frac[0] = 0
       frac ← frac[1–52] || 0b0
       exp ← exp – 1
    End
  frD[0] ← sign
  frD[1–11] ← exp + 1023
  frD[12–63] ← frac[1–52]
```

Infinity / QNaN / SNaN / Zero

```
If WORD[1–8] = 255 or WORD[1–31] = 0
  frD[0–1] ← WORD[0–1]
  frD[2] ← WORD[1]
  frD[3] ← WORD[1]
  frD[4] ← WORD[1]
  frD[5–63] ← WORD[2–31] || (29)0
```

For double-precision floating-point load instructions, no conversion is required as the data from memory is copied directly into the FPRs.

Many floating-point load instructions have an update form in which **r**A is updated with the EA. For these forms, if operand **r**A ≠ 0, the effective address (EA) is placed into register **r**A and the memory element (word or double word) addressed by the EA is loaded into the FPR specified by operand **fr**D; if operand **r**A = 0, the instruction form is invalid.

C.7 Floating-Point Store Instructions

There are three basic forms of store instruction—single-precision, double-precision, and integer. The integer form is provided by the optional **stfiwx** instruction. Because the FPRs support only floating-point double format for floating-point data, single-precision store floating-point instructions convert double-precision data to single-precision format prior to storing the operands into memory. The conversion steps follow:

Let WORD[0–31] be the word written to in memory.

No Denormalization Required (includes Zero/Infinity/NaN)

```
if frS[1–11] > 896 or frS[1–63] = 0 then
  WORD[0–1] ← frS[0–1]
  WORD[2–31] ← frS[5–34]
```

Denormalization Required

```
if 874 ≤ frS[1–11] ≤ 896 then
         sign ← frS[0]
         exp ← frS[1–11] – 1023
         frac ← 0b1 || frS[12–63]
         Denormalize operand
                  Do while exp < –126
                           frac ← 0b0 || frac[0–62]
                           exp ← exp + 1
                  End
         WORD[0] ← sign
         WORD[1–8] ← 0x00
         WORD[9–31] ← frac[1–23]
else WORD ← undefined
```

Note that if the value to be stored by a single-precision store floating-point instruction is larger in magnitude than the maximum number representable in single format, the first case mentioned, “No Denormalization Required,” applies. The result stored in WORD is then a well-defined value but is not numerically equal to the value in the source register (that is, the result of a single-precision load floating-point from WORD does not compare equal to the contents of the original source register).

Note that the description of conversion steps presented here is only a model. The actual implementation may vary from this description but must produce results equivalent to what this model would produce.

Note that for double-precision store floating-point instructions and for the store floating-point as integer word instruction, no conversion is required as the data from the FPR is copied directly into memory.

Appendix D
Synchronization Programming Examples

The examples in this appendix show how synchronization instructions can be used to emulate various synchronization primitives and how to provide more complex forms of synchronization.

For each of these examples, it is assumed that a similar sequence of instructions is used by all processes requiring synchronization of the accessed data.

D.1 General Information

The following points provide general information about the **lwarx** and **stwcx.** instructions:

- In general, **lwarx** and **stwcx.** instructions should be paired, with the same effective address (EA) used for both. The only exception is that an unpaired **stwcx.** instruction to any (scratch) effective address can be used to clear any reservation held by the processor.
- It is acceptable to execute an **lwarx** instruction for which no **stwcx.** instruction is executed. Such a dangling **lwarx** instruction occurs in the example shown in Section D.2.5, “Test and Set,” if the value loaded is not zero.
- To increase the likelihood that forward progress is made, it is important that looping on **lwarx**/**stwcx.** pairs be minimized. For example, in the sequence shown in Section D.2.5, “Test and Set,” this is achieved by testing the old value before attempting the store—were the order reversed, more **stwcx.** instructions might be executed, and reservations might more often be lost between the **lwarx** and the **stwcx.** instructions.
- The manner in which **lwarx** and **stwcx.** are communicated to other processors and mechanisms, and between levels of the memory subsystem within a given processor, is implementation-dependent. In some implementations, performance may be improved by minimizing looping on an **lwarx** instruction that fails to return a desired value. For example, in the example provided in Section D.2.5, “Test and Set,” if the program stays in the loop until the word loaded is zero, the programmer can change the “**bne**- $+12” to “**bne**- loop.”
- In some implementations, better performance may be obtained by using an ordinary load instruction to do the initial checking of the value, as follows:

```
loop:    lwz      r5,0(r3) #load the word
         cmpwi    r5,0     #loop back if word
         bne-     loop     #not equal to 0
         lwarx    r5,0,r3  #try again, reserving
         cmpwi    r5,0     #(likely to succeed)
         bne      loop     #try to store nonzero
         stwcx.   r4,0,r3  #
         bne-     loop     #loop if lost reservation
```

- In a multiprocessor, livelock (a state in which processors interact in a way such that no processor makes progress) is possible if a loop containing an **lwarx**/**stwcx.** pair also contains an ordinary

store instruction for which any byte of the affected memory area is in the reservation granule of the reservation. For example, the first code sequence shown in Section D.5, “List Insertion,” can cause livelock if two list elements have next element pointers in the same reservation granule.

D.2 Synchronization Primitives

The following examples show how the **lwarx** and **stwcx.** instructions can be used to emulate various synchronization primitives. The sequences used to emulate the various primitives consist primarily of a loop using the **lwarx** and **stwcx.** instructions. Additional synchronization is unnecessary, because the **stwcx.** will fail, clearing the EQ bit, if the word loaded by **lwarx** has changed before the **stwcx.** is executed.

D.2.1 Fetch and No-Op

The fetch and no-op primitive atomically loads the current value in a word in memory. In this example, it is assumed that the address of the word to be loaded is in GPR3 and the data loaded are returned in GPR4.

```
loop:   lwarx   r4,0,r3  #load and reserve
        stwcx.  r4,0,r3  #store old value if still reserved
        bne-    loop     #loop if lost reservation
```

The **stwcx.**, if it succeeds, stores to the destination location the same value that was loaded by the preceding **lwarx**. While the store is redundant with respect to the value in the location, its success ensures that the value loaded by the **lwarx** was the current value (that is, the source of the value loaded by the **lwarx** was the last store to the location that preceded the **stwcx.** in the coherence order for the location).

D.2.2 Fetch and Store

The fetch and store primitive atomically loads and replaces a word in memory.

In this example, it is assumed that the address of the word to be loaded and replaced is in GPR3, the new value is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3  #load and reserve
        stwcx.  r4,0,r3  #store new value if still reserved
        bne-    loop     #loop if lost reservation
```

D.2.3 Fetch and Add

The fetch and add primitive atomically increments a word in memory.

In this example, it is assumed that the address of the word to be incremented is in GPR3, the increment is in GPR4, and the old value is returned in GPR5.

```
loop:   lwarx   r5,0,r3          #load and reserve
        add     r0,r4,r5         #increment word
        stwcx.  r0,0,r3          #store new value if still reserved
        bne-    loop             #loop if lost reservation
```

D.2.4 Fetch and AND

The fetch and AND primitive atomically ANDs a value into a word in memory.

In this example, it is assumed that the address of the word to be ANDed is in GPR3, the value to AND into it is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx    r5,0,r3          #load and reserve
         and      r0,r4,r5         #AND word
         stwcx.   r0,0,r3          #store new value if still reserved
         bne-     loop             #loop if lost reservation
```

This sequence can be changed to perform another Boolean operation atomically on a word in memory, simply by changing the AND instruction to the desired Boolean instruction (OR, XOR, etc.).

D.2.5 Test and Set

This version of the test and set primitive atomically loads a word from memory, ensures that the word in memory is a nonzero value, and sets CR0[EQ] according to whether the value loaded is zero.

In this example, it is assumed that the address of the word to be tested is in GPR3, the new value (nonzero) is in GPR4, and the old value is returned in GPR5.

```
loop:    lwarx    r5,0,r3  #load and reserve
         cmpwi    r5, 0    #done if word
         bne      $+12     #not equal to 0
         stwcx.   r4,0,r3  #try to store non-zero
         bne-     loop     #loop if lost reservation
```

D.3 Compare and Swap

The compare and swap primitive atomically compares a value in a register with a word in memory. If they are equal, it stores the value from a second register into the word in memory. If they are unequal, it loads the word from memory into the first register, and sets the EQ bit of the CR0 field to indicate the result of the comparison.

In this example, it is assumed that the address of the word to be tested is in GPR3, the word that is compared is in GPR4, the new value is in GPR5, and the old value is returned in GPR4.

```
loop:    lwarx    r6,0,r3  #load and reserve
         cmpw     r4,r6    #first 2 operands equal ?
         bne-     exit     #skip if not
         stwcx.   r5,0,r3  #store new value if still reserved
         bne-     loop     #loop if lost reservation
exit:    mr       r4,r6    #return value from memory
```

Notes:

- The semantics in this example are based on the IBM System/370™ compare and swap instruction. Other architectures may define this instruction differently.
- Compare and swap is shown primarily for pedagogical reasons. It is useful on machines that lack the better synchronization facilities provided by the **lwarx** and **stwcx.** instructions. Although the instruction is atomic, it checks only for whether the current value matches the old value. An error can occur if the value had been changed and restored before being tested.

- In some applications, the second **bne-** instruction and/or the **mr** instruction can be omitted. The first **bne-** is needed only if the application requires that if the EQ bit of CR0 field on exit indicates not equal, then the original compared value in **r**4 and **r**6 are in fact not equal. The **mr** is needed only if the application requires that if the compared values are not equal, then the word from memory is loaded into the register with which it was compared (rather than into a third register). If either, or both, of these instructions is omitted, the resulting compare and swap does not obey the IBM System/370 semantics.

D.4 Lock Acquisition and Release

This example provides an algorithm for locking that demonstrates the use of synchronization with an atomic read/modify/write operation. GPR3 provides a shared memory location, the address of which is an argument of the lock and unlock procedures. This argument is used as a lock to control access to some shared resource such as a data structure. The lock is open when its value is zero and locked when it is one. Before accessing the shared resource, a processor sets the lock by having the lock procedure call TEST_AND_SET, which executes the code sequence in Section D.2.5, “Test and Set.” This atomically sets the old value of the lock, and writes the new value (1) given to it in GPR4, returning the old value in GPR5 (not used in the following example) and setting the EQ bit in CR0 according to whether the value loaded is zero. The lock procedure repeats the test and set procedure until it successfully changes the value in the lock from zero to one.

The processor must not access the shared resource until it sets the lock. After the **bne-** instruction that checks for the successful test and set operation, the processor executes the **isync** instruction. This delays all subsequent instructions until all previous instructions have completed to the extent required by context synchronization. The **sync** instruction could be used but performance would be degraded because the **sync** instruction waits for all outstanding memory accesses to complete with respect to other processors. This is not necessary here.

```
lock:   li      r4,1            #obtain lock
loop:   bl      test_and_set    #test and set
        bne-    loop            #retry until old = 0
                                #delay subsequent instructions until
                                #previous ones complete
        isync
        blr                     #return
```

The unlock procedure writes a zero to the lock location. If the access to the shared resource includes write operations, most applications that use locking require the processor to execute a **sync** instruction to make its modification visible to all processors before releasing the lock. For this reason, the unlock procedure in the following example begins with a **sync**.

```
unlock: sync                    #delay until prior stores finish
        li      r1,0
        stw     r1,0(r3)        #store zero to lock location
        blr                     #return
```

D.5 List Insertion

The following example shows how the **lwarx** and **stwcx.** instructions can be used to implement simple LIFO (last-in-first-out) insertion into a singly-linked list. (Complicated list insertion, in which multiple values must be changed atomically, or in which the correct order of insertion depends on the contents of the elements, cannot be implemented in the manner shown below, and requires a more complicated strategy such as using locks.)

The next element pointer from the list element after which the new element is to be inserted, here called the parent element, is stored into the new element, so that the new element points to the next element in the list—this store is performed unconditionally. Then the address of the new element is conditionally stored into the parent element, thereby adding the new element to the list.

In this example, it is assumed that the address of the parent element is in GPR3, the address of the new element is in GPR4, and the next element pointer is at offset zero from the start of the element. It is also assumed that the next element pointer of each list element is in a reservation granule separate from that of the next element pointer of all other list elements.

```
loop:    lwarx    r2,0,r3  #get next pointer
         stw      r2,0(r4) #store in new element
         sync              #let store settle (can omit if not MP)
         stwcx.   r4,0,r3  #add new element to list
         bne-     loop     #loop if stwcx. failed
```

In the preceding example, if two list elements have next element pointers in the same reservation granule in a multiprocessor system, livelock can occur.

If it is not possible to allocate list elements such that each element’s next element pointer is in a different reservation granule, livelock can be avoided by using the following sequence:

```
         lwz      r2,0(r3) #get next pointer
loopl:   mr       r5,r2    #keep a copy
         stw      r2,0(r4) #store in new element
         sync              #let store settle
loop2:   lwarx    r2,0,r3  #get it again
         cmpw     r2,r5    #loop if changed (someone
         bne-     loopl    #else progressed)
         stwcx.   r4,0,r3  #add new element to list
         bne-     loop2    #loop if failed
```

Appendix E
Simplified Mnemonics for PowerPC Instructions

This chapter describes simplified mnemonics, which are provided for easier coding of assembly language programs. Simplified mnemonics are defined for the most frequently used forms of branch conditional, compare, trap, rotate and shift, and certain other instructions defined by the PowerPC™ architecture and by implementations of and extensions to the PowerPC architecture.

Most of this information is also provided in the appendixes of reference manuals and the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (referred to as the *Programming Environment Manual*). However, Section E.10, "Comprehensive List of Simplified Mnemonics," provides an alphabetical listing of simplified mnemonics that are used by a variety of processors. Some assemblers may define additional simplified mnemonics not included here. The simplified mnemonics listed here should be supported by all compilers.

E.1 Overview

Simplified (or extended) mnemonics allow an assembly-language programmer to program using more intuitive mnemonics and symbols than the instructions and syntax defined by the instruction set architecture. For example, to code the conditional call "branch to an absolute target if CR4 specifies a greater than condition, setting the LR without simplified mnemonics, the programmer would write the branch conditional instruction, **bc 12,17,***target*. The simplified mnemonic, branch if greater than, **bgt cr4,***target*, incorporates the conditions. Not only is it easier to remember the symbols than the numbers when programming, it is also easier to interpret simplified mnemonics when reading existing code.

Although the original PowerPC architecture documents include a set of simplified mnemonics, these are not a formal part of the architecture, but rather a recommendation for assemblers that support the instruction set.

Many simplified mnemonics have been added to those originally included in the architecture documentation. Some assemblers created their own, and others have been added to support extensions to the instruction set (for example, AltiVec instructions and Book E auxiliary processing units (APUs)). Simplified mnemonics have been added for new architecturally defined and new implementation-specific special-purpose registers (SPRs). These simplified mnemonics are described only in a very general way.

E.2 Subtract Simplified Mnemonics

This section describes simplified mnemonics for subtract instructions.

E.2.1 Subtract Immediate

There is no subtract immediate instruction, however, its effect is achieved by negating the immediate operand of an Add Immediate instruction, **addi**. Simplified mnemonics include this negation, making the intent of the computation more clear. These are listed in Table E-1.

Table E-1. Subtract Immediate Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic
subi r**D**,r**A**,value	**addi** r**D**,r**A**,–value
subis r**D**,r**A**,value	**addis** r**D**,r**A**,–value
subic r**D**,r**A**,value	**addic** r**D**,r**A**,–value
subic. r**D**,r**A**,value	**addic.** r**D**,r**A**,–value

E.2.2 Subtract

Subtract from instructions subtract the second operand (**r**A) from the third (**r**B). The simplified mnemonics in Table E-2 use the more common order in which the third operand is subtracted from the second.

Table E-2. Subtract Simplified Mnemonics

Simplified Mnemonic	Standard Mnemonic[1]
sub[**o**][.] r**D**,r**A**,r**B**	**subf**[**o**][.] r**D**,r**B**,r**A**
subc[**o**][.] r**D**,r**A**,r**B**	**subfc**[**o**][.] r**D**,r**B**,r**A**

[1] r**D**,r**B**,r**A** is not the standard order for the operands. The order of r**B** and r**A** is reversed to show the equivalent behavior of the simplified mnemonic.

E.3 Rotate and Shift Simplified Mnemonics

Rotate and shift instructions provide powerful, general ways to manipulate register contents, but can be difficult to understand. Simplified mnemonics are provided for the following operations:

- Extract—Select a field of *n* bits starting at bit position *b* in the source register; left or right justify this field in the target register; clear all other bits of the target register.
- Insert—Select a left- or right-justified field of *n* bits in the source register; insert this field starting at bit position *b* of the target register; leave other bits of the target register unchanged.
- Rotate—Rotate the contents of a register right or left *n* bits without masking.
- Shift—Shift the contents of a register right or left *n* bits, clearing vacated bits (logical shift).
- Clear—Clear the leftmost or rightmost *n* bits of a register.
- Clear left and shift left—Clear the leftmost *b* bits of a register, then shift the register left by *n* bits. This operation can be used to scale a (known non-negative) array index by the width of an element.

E.3.1 Operations on Words

The simplified mnemonics in Table E-3 can be coded with a dot (.) suffix to cause the Rc bit to be set in the underlying instruction.

Table E-3. Word Rotate and Shift Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Extract and left justify word immediate	**extlwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwinm** r**A**,r**S**,*b*,**0**,*n* – 1
Extract and right justify word immediate	**extrwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwinm** r**A**,r**S**,*b* + *n*, 32 – *n*,**31**
Insert from left word immediate	**inslwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwimi** r**A**,r**S**,32 – *b*,*b*,(*b* + *n*) – 1
Insert from right word immediate	**insrwi** r**A**,r**S**,*n*,*b* (*n* > 0)	**rlwimi** r**A**,r**S**,32 – (*b* + *n*),*b*,(*b* + *n*) – 1
Rotate left word immediate	**rotlwi** r**A**,r**S**,*n*	**rlwinm** r**A**,r**S**,*n*,**0**,**31**
Rotate right word immediate	**rotrwi** r**A**,r**S**,*n*	**rlwinm** r**A**,r**S**,32 – *n*,**0**,**31**
Rotate word left	**rotlw** r**A**,r**S**,r**B**	**rlwnm** r**A**,r**S**,r**B**,**0**,**31**
Shift left word immediate	**slwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,*n*,**0**,31 – *n*
Shift right word immediate	**srwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,32 – *n*,*n*,**31**
Clear left word immediate	**clrlwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,**0**,*n*,**31**
Clear right word immediate	**clrrwi** r**A**,r**S**,*n* (*n* < 32)	**rlwinm** r**A**,r**S**,**0**,**0**,31 – *n*
Clear left and shift left word immediate	**clrlslwi** r**A**,r**S**,*b*,*n* (*n* ≤ *b* ≤ 31)	**rlwinm** r**A**,r**S**,*n*,*b* – *n*,31 – *n*

Examples using word mnemonics follow:

1. Extract the sign bit (bit 0) of **r**S and place the result right-justified into **r**A.
   extrwi rA,**r**S,**1,0** equivalent to **rlwinm r**A,**r**S,**1,31,31**
2. Insert the bit extracted in (1) into the sign bit (bit 0) of **r**B.
   insrwi rB,**r**A,**1,0** equivalent to **rlwimi r**B,**r**A,**31,0,0**
3. Shift the contents of **r**A left 8 bits.
   slwi rA,**r**A,**8** equivalent to **rlwinm r**A,**r**A,**8,0,23**
4. Clear the high-order 16 bits of **r**S and place the result into **r**A.
   clrlwi rA,**r**S,**16** equivalent to **rlwinm r**A,**r**S,**0,16,31**

E.4 Branch Instruction Simplified Mnemonics

Branch conditional instructions can be coded with the operations, a condition to be tested, and a prediction, as part of the instruction mnemonic rather than as numeric operands (the BO and BI operands). Table E-4 shows the four general types of branch instructions. Simplified mnemonics are defined only for branch instructions that include BO and BI operands; there is no need to simplify unconditional branch mnemonics.

Table E-4. Branch Instructions

Instruction Name	Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI

The BO and BI operands correspond to two fields in the instruction opcode, as Figure E-1 shows for Branch Conditional (**bc**, **bca**, **bcl**, and **bcla**) instructions.

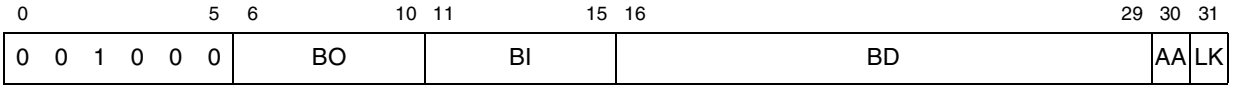


Figure E-1. Branch Conditional (bc) Instruction Format

The BO operand specifies branch operations that involve decrementing CTR. It is also used to determine whether testing a CR bit causes a branch to occur if the condition is true or false.

The BI operand identifies a CR bit to test (whether a comparison is less than or greater than, for example). The simplified mnemonics avoid the need to memorize the numerical values for BO and BI.

For example, **bc 16,0,***target* is a conditional branch that, as a BO value of 16 (0b1_0000) indicates, decrements the CTR, then branches if the decremented CTR is not zero. The operation specified by BO is abbreviated as **d** (for decrement) and **nz** (for not zero), which replace the **c** in the original mnemonic; so the simplified mnemonic for **bc** becomes **bdnz**. The branch does not depend on a condition in the CR, so BI can be eliminated, reducing the expression to **bdnz** *target*.

In addition to CTR operations, the BO operand provides an optional prediction bit and a true or false indicator can be added. For example, if the previous instruction should branch only on an equal condition in CR0, the instruction becomes **bc 8,2,***target*. To incorporate a true condition, the BO value becomes 8 (as shown in Table E-6); the CR0 equal field is indicated by a BI value of 2 (as shown in Table E-7). Incorporating the branch-if-true condition adds a '**t**' to the simplified mnemonic, **bdnzt.** The equal condition that is specified by a BI value of 2 (indicating the EQ bit in CR0) is replaced by the **eq** symbol. Using the simplified mnemonic and the **eq** operand, the expression becomes **bdnzt eq,***target*.

This example tests CR0[EQ]; however, to test the equal condition in CR5 (CR bit 22), the expression becomes **bc 8,22,***target*. The BI operand of 22 indicates CR[22] (CR5[2], or BI field 0b10110), as shown in Table E-7. This can be expressed as the simplified mnemonic. **bdnzt 4 * cr5 + eq,***target*.

The notation, **4 * cr5 + eq** may at first seem awkward, but it eliminates computing the value of the CR bit. It can be seen that (4 * 5) + 2 = 22.

E.4.1 Key Facts about Simplified Branch Mnemonics

The following key points are helpful in understanding how to use simplified branch mnemonics:

- All simplified branch mnemonics eliminate the BO operand, so if any operand is present in a branch simplified mnemonic, it is the BI operand (or a reduced form of it).
- If the CR is not involved in the branch, the BI operand can be deleted
- If the CR is involved in the branch, the BI operand can be treated in the following ways:
  - It can be specified as a numeric value, just as it is in the architecturally defined instruction, or it can be indicated with an easier to remember formula, **4** * **cr***n* + [test bit symbol], where *n* indicates the CR field number.
  - The condition of the test bit (eq, lt, gt, and so) can be incorporated into the mnemonic, leaving the need for an operand that defines only the CR field.
    - If the test bit is in CR0, no operand is needed.
    - If the test bit is in CR1–CR7, the BI operand can be replaced with a **cr**S operand (that is, **cr1**, **cr2**, **cr3**, and so forth.

E.4.2 Eliminating the BO Operand

The 5-bit BO field, shown in Figure E-2, encodes the following operations in conditional branch instructions:

- Decrement count register (CTR)
  - And test if result is equal to zero
  - And test if result is not equal to zero
- Test condition register (CR)
  - Test condition true
  - Test condition false
- Branch prediction (taken, fall through). If the prediction bit, *y*, is needed, it is signified by appending a plus or minus sign as described in Section E.4.3, “Incorporating the BO Branch Prediction.”

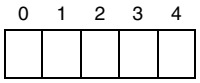


Figure E-2. BO Field (Bits 6–10 of the Instruction Encoding)

BO bits can be interpreted individually as described in Table E-5.

Table E-5. BO Bit Encodings

BO Bit	Description
0	If set, ignore the CR bit comparison.
1	If set, the CR bit comparison is against true, if not set the CR bit comparison is against false
2	If set, the CTR is not decremented.
3	If BO[2] is set, this bit determines whether the CTR comparison is for equal to zero or not equal to zero.
4	The *y* bit. If set, reverse the static prediction. Use of the this bit is optional and independent from the interpretation of the rest of the BO operand. Because simplified branch mnemonics eliminate the BO operand, this bit is programmed by adding a plus or minus sign to the simplified mnemonic, as described in Section E.4.3, "Incorporating the BO Branch Prediction."

Thus, a BO encoding of 10100 (decimal 20) means ignore the CR bit comparison and do not decrement the CTR—in other words, branch unconditionally. Encodings for the BO operand are shown in Table E-6. A *z* bit indicates that the bit is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

As shown in Table E-6, the '**c**' in the standard mnemonic is replaced with the operations otherwise specified in the BO field, (**d** for decrement, **z** for zero, **nz** for non-zero, **t** for true, and **f** for false).

Table E-6. BO Operand Encodings

BO Field	Value[1] (Decimal)	Description	Symbol
0000*y*	0	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is FALSE.	**dnzf**
0001*y*	2	Decrement the CTR, then branch if the decremented CTR = 0; condition is FALSE.	**dzf**
001*zy*	4	Branch if the condition is FALSE.[2] Note that 'false' and 'four' both start with 'f'.	**f**
0100*y*	8	Decrement the CTR, then branch if the decremented CTR ≠ 0; condition is TRUE.	**dnzt**
0101*y*	10	Decrement the CTR, then branch if the decremented CTR = 0; condition is TRUE.	**dzt**
011*z*[3]*y*	12	Branch if the condition is TRUE. [2] Note that 'true' and 'twelve' both start with 't'.	**t**
1*z*00*y* [4]	16	Decrement the CTR, then branch if the decremented CTR ≠ 0.	**dnz**[5]
1*z*01*y* [4]	18	Decrement the CTR, then branch if the decremented CTR = 0.	**dz** [5]
1*z*1*zz* [4]	20	Branch always.	—

[1] Assumes y = z = 0. Section E.4.3, "Incorporating the BO Branch Prediction," describes how to use simplified mnemonics to program the *y* bit for static prediction.

[2] Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section E.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

[3] A *z* bit indicates a bit that is ignored. However, these bits should be cleared, as they may be assigned a meaning in a future version of the architecture.

[4] Simplified mnemonics for branch instructions that do not test CR bits (BO = 16, 18, and 20) should specify only a target. Otherwise a programming error may occur.

[5] Notice that these instructions do not use the branch if condition true or false operations. For that reason, simplified mnemonics for these should not specify a BI operand.

E.4.3 Incorporating the BO Branch Prediction

As shown in Table E-6, the low-order bit (*y* bit) of the BO field provides a hint about whether the branch is likely to be taken (static branch prediction). Assemblers should clear this bit unless otherwise directed. This default action indicates the following:

- A branch conditional with a negative displacement field is predicted to be taken.
- A branch conditional with a non-negative displacement field is predicted not to be taken (fall through).
- A branch conditional to an address in the LR or CTR is predicted not to be taken (fall through).

If the likely outcome (branch or fall through) of a given branch conditional instruction is known, a suffix can be added to the mnemonic that tells the assembler how to set the *y* bit. That is, '+' indicates that the branch is to be taken and '–' indicates that the branch is not to be taken. This suffix can be added to any branch conditional mnemonic, either standard or simplified.

For relative and absolute branches (**bc**[**l**][**a**]), the setting of the *y* bit depends on whether the displacement field is negative or non-negative. For negative displacement fields, coding the suffix '+' causes the bit to be cleared, and coding the suffix '–' causes the bit to be set. For non-negative displacement fields, coding the suffix '+' causes the bit to be set, and coding the suffix '–' causes the bit to be cleared.

For branches to an address in the LR or CTR (**bclr**[**l**] or **bcctr**[**l**]), coding the suffix '+' causes the *y* bit to be set, and coding the suffix '–' causes the bit to be cleared.

Examples of branch prediction follow:

1. Branch if CR0 reflects less than condition, specifying that the branch should be predicted as taken.
   blt+ *target*
2. Same as (1), but target address is in the LR and the branch should be predicted as not taken.
   bltlr–

E.4.4 The BI Operand—CR Bit and Field Representations

With standard branch mnemonics, the BI operand is used when it is necessary to test a CR bit, as shown in the example in Section E.4, "Branch Instruction Simplified Mnemonics,"

With simplified mnemonics, the BI operand is handled differently depending on whether the simplified mnemonic incorporates a CR condition to test, as follows:

- Some branch simplified mnemonics incorporate only the BO operand. These simplified mnemonics can use the architecturally defined BI operand to specify the CR bit, as follows:
  - The BI operand can be presented exactly as it is with standard mnemonics—as a decimal number, 0–31.
  - Symbols can be used to replace the decimal operand, as shown in the example in Section E.4, "Branch Instruction Simplified Mnemonics," where **bdnzt 4 * cr5 + eq,***target* could be used instead of **bdnzt 22,***target*. This is described in Section E.4.4.1.1, "Specifying a CR Bit."

  The simplified mnemonics in Section E.4.5, "Simplified Mnemonics that Incorporate the BO Operand," use one of these two methods to specify a CR bit.

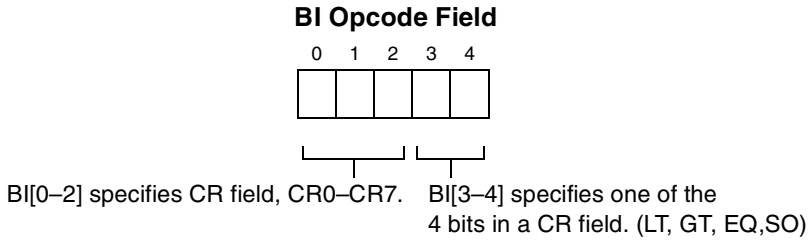
- Additional simplified mnemonics are specified that incorporate CR conditions that would otherwise be specified by the BI operand, so the BI operand is replaced by the **cr**S operand to specify the CR field, CR0–CR7. See Section E.4.4.1, “BI Operand Instruction Encoding.”

  These mnemonics are described in Section E.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS).”

E.4.4.1 BI Operand Instruction Encoding

The entire 5-bit BI field, shown in Figure E-3, represents the bit number for the CR bit to be tested. For standard branch mnemonics and for branch simplified mnemonics that do not incorporate a CR condition, the BI operand provides all 5 bits.

For simplified branch mnemonics described in Section E.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS),” the BI operand is replaced by a **cr**S operand. To understand this, it is useful to view the BI operand as comprised of two parts. As Figure E-3 shows, BI[0–2] indicates the CR field and BI[3–4] represents the condition to test.



Simplified mnemonics based on CR conditions but not CTR values—BO = 12 (branch if true) and BO = 4 branch if false)	Specified by a separate, reduced BI operand (**cr**S)	Incorporated into the simplified mnemonic.
Standard branch mnemonics and simplified mnemonics based on CTR values	The BI operand specifies the entire 5-bit field. If CR0 is used, the bit can be identified by LT, GT, EQ, or SO. If CR1–CR7 are used, the form 4 * **cr**S + LT|GT|EQ|SO can be used.	

Figure E-3. BI Field (Bits 11–14 of the Instruction Encoding)

Integer record-form instructions update CR0, as described in Table E-7.

E.4.4.1.1 Specifying a CR Bit

Note that the AIM version the PowerPC architecture numbers CR bits 0–31 and Book E numbers them 32–63. However, no adjustment is necessary to the code; in Book E devices, 32 is automatically added to the BI value, as shown in Table E-7 and Table E-8.

Table E-7. CR0 and CR1 Fields as Updated by Integer Instructions

CR*n* Bit	CR Bits		BI		Description
	AIM	Book E	0–2	3–4	
CR0[0]	0	32	000	00	Negative (LT)—Set when the result is negative.
CR0[1]	1	33	000	01	Positive (GT)—Set when the result is positive (and not zero).

Table E-7. CR0 and CR1 Fields as Updated by Integer Instructions (continued)

CR*n* Bit	CR Bits		BI		Description
	AIM	Book E	0–2	3–4	
CR0[2]	2	34	000	10	Zero (EQ)—Set when the result is zero.
CR0[3]	3	35	000	11	Summary overflow (SO). Copy of XER[SO] at the instruction's completion.

Some simplified mnemonics incorporate only the BO field (as described Section E.4.2, "Eliminating the BO Operand"). If one of these simplified mnemonics is used and the CR must be accessed, the BI operand can be specified either as a numeric value or by using the symbols in Table E-8.

Compare word instructions (described in Section E.5, "Compare Word Simplified Mnemonics"), move to CR instructions, and others can also modify CR fields, so CR0 and CR1 may hold values that do not adhere to the meanings described in Table E-7. CR logical instructions, described in Section E.6, "Condition Register Logical Simplified Mnemonics," can update individual CR bits.

Table E-8. BI Operand Settings for CR Fields for Branch Comparisons

CR*n* Bit	Bit Expression	CR Bits		BI		Description
		AIM (BI Operand)	Book E	0–2	3–4	
CR*n*[0]	**4 * cr0 + lt** (or **lt**) **4 * cr1 + lt** **4 * cr2 + lt** **4 * cr3+ lt** **4 * cr4 + lt** **4 * cr5 + lt** **4 * cr6 + lt** **4 * cr7 + lt**	0 4 8 12 16 20 24 28	32 36 40 44 48 52 56 60	000 001 010 011 100 101 110 111	00	Less than (LT). For integer compare instructions: **r**A < SIMM or **r**B (signed comparison) or **r**A < UIMM or **r**B (unsigned comparison).
CR*n*[1]	**4 * cr0 + gt** (or **gt**) **4 * cr1 + gt** **4 * cr2 + gt** **4 * cr3+ gt** **4 * cr4 + gt** **4 * cr5 + gt** **4 * cr6 + gt** **4 * cr7 + gt**	1 5 9 13 17 21 25 29	33 37 41 45 49 53 57 61	000 001 010 011 100 101 110 111	01	Greater than (GT). For integer compare instructions: rA > SIMM or rB (signed comparison) or **r**A > UIMM or **r**B (unsigned comparison).
CR*n*[2]	**4 * cr0 + eq** (or **eq**) **4 * cr1 + eq** **4 * cr2 + eq** **4 * cr3+ eq** **4 * cr4 + eq** **4 * cr5 + eq** **4 * cr6 + eq** **4 * cr7 + eq**	2 6 10 14 18 22 26 30	34 38 42 46 50 54 58 62	000 001 010 011 100 101 110 111	10	Equal (EQ). For integer compare instructions: **r**A = SIMM, UIMM, or **r**B.

Table E-8. BI Operand Settings for CR Fields for Branch Comparisons (continued)

CR*n* Bit	Bit Expression	CR Bits		BI		Description
		AIM (BI Operand)	Book E	0–2	3–4	
CR*n*[3]	**4 * cr0 + so/un** (or **so/un**) **4 * cr1 + so/un** **4 * cr2 + so/un** **4 * cr3 + so/un** **4 * cr4 + so/un** **4 * cr5 + so/un** **4 * cr6 + so/un** **4 * cr7 + so/un**	3 7 11 15 19 23 27 31	35 39 43 47 51 55 59 63	000 001 010 011 100 101 110 111	11	Summary overflow (SO). For integer compare instructions, this is a copy of XER[SO] at instruction completion.

To provide simplified mnemonics for every possible combination of BO and BI (that is, including bits that identified the CR field) would require 2^{10} = 1024 mnemonics, most of which would be only marginally useful. The abbreviated set in Section E.4.5, "Simplified Mnemonics that Incorporate the BO Operand," covers useful cases. Unusual cases can be coded using a standard branch conditional syntax.

E.4.4.1.2 The crS Operand

The **cr**S symbols are shown in Table E-9. Note that either the symbol or the operand value can be used in the syntax used with the simplified mnemonic.

Table E-9. CR Field Identification Symbols

Symbol	BI[0–2]	CR Bits
cr0 (default, can be eliminated from syntax)	000	32–35
cr1	001	36–39
cr2	010	40–43
cr3	011	44–47
cr4	100	48–51
cr5	101	52–55
cr6	110	56–59
cr7	111	60–63

To identify a CR bit, an expression in which a CR field symbol is multiplied by 4 and then added to a bit-number-within-CR-field symbol can be used, (for example, **cr0 * 4 + eq**).

E.4.5 Simplified Mnemonics that Incorporate the BO Operand

The mnemonics in Table E-10 allow common BO operand encodings to be specified as part of the mnemonic, along with the absolute address (AA) and set link register bits (LK). There are no simplified mnemonics for relative and absolute unconditional branches. For these, the basic mnemonics **b**, **ba**, **bl**, and **bla** are used.

Table E-10. Branch Simplified Mnemonics

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Branch unconditionally [1]	—	—	**blr**	**bctr**	—	—	**blrl**	**bctrl**
Branch if condition true	**bt**	**bta**	**btlr**	**btctr**	**btl**	**btla**	**btlrl**	**btctrl**
Branch if condition false	**bf**	**bfa**	**bflr**	**bfctr**	**bfl**	**bfla**	**bflrl**	**bfctrl**
Decrement CTR, branch if CTR ≠ 0 [1]	**bdnz**	**bdnza**	**bdnzlr**	—	**bdnzl**	**bdnzla**	**bdnzlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bdnzt**	**bdnzta**	**bdnztlr**	—	**bdnztl**	**bdnztla**	**bdnztlrl**	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bdnzf**	**bdnzfa**	**bdnzflr**	—	**bdnzfl**	**bdnzfla**	**bdnzflrl**	—
Decrement CTR, branch if CTR = 0 [1]	**bdz**	**bdza**	**bdzlr**	—	**bdzl**	**bdzla**	**bdzlrl**	—
Decrement CTR, branch if CTR = 0 and condition true	**bdzt**	**bdzta**	**bdztlr**	—	**bdztl**	**bdztla**	**bdztlrl**	—
Decrement CTR, branch if CTR = 0 and condition false	**bdzf**	**bdzfa**	**bdzflr**	—	**bdzfl**	**bdzfla**	**bdzflrl**	—

[1] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table E-10 shows the syntax for basic simplified branch mnemonics

Table E-11. Branch Instructions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	N/A, syntax does not include BO	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x* [1](**b***x***a b***x***l b***x***la**)	BI[2],target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b***x***lr** (**b***x***lrl**)	BI
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b***x***ctr** (**b***x***ctrl**)	BI

[1] *x* stands for one of the symbols in Table E-6, where applicable.

[2] BI can be a numeric value or an expression as shown in Table E-9.

The simplified mnemonics in Table E-10 that test a condition require a corresponding CR bit as the first operand (as examples 2–5 below illustrate). The symbols in Table E-9 can be used in place of a numeric value.

E.4.5.1 Examples that Eliminate the BO Operand

The simplified mnemonics in Table E-10 are used in the following examples:

1. Decrement CTR and branch if it is still nonzero (closure of a loop controlled by a count loaded into CTR) (note that no CR bits are tested).
   bdnz *target* equivalent to **bc 16,0,***target*
   Because this instruction does not test a CR bit, the simplified mnemonic should specify only a target operand. Specifying a CR (for example, **bdnz** 0,*target* or **bdnz cr0,***target*) may be considered a programming error. Subsequent examples test conditions).
2. Same as (1) but branch only if CTR is nonzero and equal condition in CR0.
   bdnzt eq,*target* equivalent to **bc 8,2,***target*
   Other equivalents include **bdnzt 2,***target* or the unlikely **bdnzt 4*cr0+eq,***target*
3. Same as (2), but equal condition is in CR5.
   bdnzt 4 * cr5 + eq,*target* equivalent to **bc 8,22,***target*
   bdnzt 22,*target* would also work
4. Branch if bit 59 of CR is false.
   bf 27,*target* equivalent to **bc 4,27,***target*
   bf 4*cr6+so,*target* would also work
5. Same as (4), but set the link register. This is a form of conditional call.
   bfl 27,*target* equivalent to **bcl 4,27,***target*

Table E-12 lists simplified mnemonics and syntax for **bc** and **bca** without LR updating.

Table E-12. Simplified Mnemonics for bc and bca without LR Update

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true[1]	**bc 12,BI,**target	**bt BI,**target	**bca 12,BI,**target	**bta BI,**target
Branch if condition false [1]	**bc 4,BI,**target	**bf BI,**target	**bca 4,BI,**target	**bfa BI,**target
Decrement CTR, branch if CTR ≠ 0	**bc 16,0,**target	**bdnz** target [2]	**bca 16,0,**target	**bdnza** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bc 8,BI,**target	**bdnzt BI,**target	**bca 8,BI,**target	**bdnzta BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bc 0,BI,**target	**bdnzf BI,**target	**bca 0,BI,**target	**bdnzfa BI,**target
Decrement CTR, branch if CTR = 0	**bc 18,0,**target	**bdz** target[2]	**bca 18,0,**target	**bdza** target[2]
Decrement CTR, branch if CTR = 0 and condition true	**bc 10,BI,**target	**bdzt BI,**target	**bca 10,BI,**target	**bdzta BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bc 2,BI,**target	**bdzf BI,**target	**bca 2,BI,**target	**bdzfa BI,**target

[1] Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section E.4.6, "Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)."

[2] Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. Otherwise a programming error may occur.

Table E-13 lists simplified mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table E-13. Simplified Mnemonics for bclr and bcctr without LR Update

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Branch unconditionally	**bclr 20,0**	**blr** [1]	**bcctr 20,0**	**bctr** [1]
Branch if condition true [2]	**bclr 12,BI**	**btlr BI**	**bcctr 12,BI**	**btctr BI**
Branch if condition false [2]	**bclr 4,BI**	**bflr BI**	**bcctr 4,BI**	**bfctr BI**
Decrement CTR, branch if CTR ≠ 0	**bclr 16,BI**	**bdnzlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclr 0,BI**	**bdnzflr BI**	—	—
Decrement CTR, branch if CTR = 0	**bclr 18,0**	**bdzlr** [1]	—	—
Decrement CTR, branch if CTR = 0 and condition true	**bclr 8,BI**	**bdnztlr BI**	—	—
Decrement CTR, branch if CTR = 0 and condition false	**bclr 2,BI**	**bdzflr BI**	—	—

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.

2 Instructions for which B0 is 12 (branch if condition true) or 4 (branch if condition false) do not depend on a CTR value and can be alternately coded by incorporating the condition specified by the BI field. See Section E.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS).”

Table E-14 provides simplified mnemonics and syntax for **bcl** and **bcla**.

Table E-14. Simplified Mnemonics for bcl and bcla with LR Update

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch unconditionally	—	—	—	—
Branch if condition true [1]	**bcl 12,BI,**target	**btl BI,**target	**bcla 12,BI,**target	**btla BI,**target
Branch if condition false [1]	**bcl 4,BI,**target	**bfl BI,**target	**bcla 4,BI,**target	**bfla BI,**target
Decrement CTR, branch if CTR ≠ 0	**bcl 16,0,**target	**bdnzl** target [2]	**bcla 16,0,**target	**bdnzla** target [2]
Decrement CTR, branch if CTR ≠ 0 and condition true	**bcl 8,0,**target	**bdnztl BI,**target	**bcla 8,BI,**target	**bdnztla BI,**target
Decrement CTR, branch if CTR ≠ 0 and condition false	**bcl 0,BI,**target	**bdnzfl BI,**target	**bcla 0,BI,**target	**bdnzfla BI,**target
Decrement CTR, branch if CTR = 0	**bcl 18,BI,**target	**bdzl** target [2]	**bcla 18,BI,**target	**bdzla** target [2]
Decrement CTR, branch if CTR = 0 and condition true	**bcl 10,BI,**target	**bdztl BI,**target	**bcla 10,BI,**target	**bdztla BI,**target
Decrement CTR, branch if CTR = 0 and condition false	**bcl 2,BI,**target	**bdzfl BI,**target	**bcla 2,BI,**target	**bdzfla BI,**target

1 Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field. See Section E.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS).”

2 Simplified mnemonics for branch instructions that do not test CR bits should specify only a target. A programming error may occur.

Table E-15 provides simplified mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table E-15. Simplified Mnemonics for bclrl and bcctrl with LR Update

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch unconditionally	**bclrl 20,0**	**blrl** [1]	**bcctrl 20,0**	**bctrl** [1]
Branch if condition true	**bclrl 12,BI**	**btlrl BI**	**bcctrl 12,BI**	**btctrl BI**
Branch if condition false	**bclrl 4,BI**	**bflrl BI**	**bcctrl 4,BI**	**bfctrl BI**
Decrement CTR, branch if CTR ≠ 0	**bclrl 16,0**	**bdnzlrl** [1]	—	—
Decrement CTR, branch if CTR ≠ 0 and condition true	**bclrl 8,BI**	**bdnztlrl BI**	—	—
Decrement CTR, branch if CTR ≠ 0 and condition false	**bclrl 0,BI**	**bdnzflrl BI**	—	—
Decrement CTR, branch if CTR = 0	**bclrl 18,0**	**bdzlrl** [1]	—	—
Decrement CTR, branch if CTR = 0 and condition true	**bclrl 10, BI**	**bdztlrl BI**	—	—
Decrement CTR, branch if CTR = 0 and condition false	**bclrl 2,BI**	**bdzflrl BI**	—	—

[1] Simplified mnemonics for branch instructions that do not test a CR bit should not specify one. A programming error may occur.

E.4.6 Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS)

The mnemonics in Table E-18 are variations of the branch-if-condition-true (BO = 12) and branch-if-condition-false (BO = 4) encodings. Because these instructions do not depend on the CTR, the true/false conditions specified by BO can be combined with the CR test bit specified by BI to create a different set of simplified mnemonics that eliminates the BO operand and the portion of the BI operand (BI[3–4]) that specifies one of the four possible test bits. However, the simplified mnemonic cannot specify in which of the eight CR fields the test bit falls, so the BI operand is replaced by a **cr**S operand.

The standard codes shown in Table E-16 are used for the most common combinations of branch conditions. Note that for ease of programming, these codes include synonyms; for example, less than or equal (**le**) and not greater than (**ng**) achieve the same result.

NOTE

A CR field symbol, **cr0**–**cr7**, is used as the first operand after the simplified mnemonic. If the default, CR0, is used, no **cr**S is necessary,

Table E-16. Standard Coding for Branch Conditions

Code	Description	Equivalent	Bit Tested
lt	Less than	—	LT
le	Less than or equal (equivalent to **ng**)	**ng**	GT
eq	Equal	—	EQ
ge	Greater than or equal (equivalent to **nl**)	**nl**	LT

Table E-16. Standard Coding for Branch Conditions (continued)

Code	Description	Equivalent	Bit Tested
gt	Greater than	—	GT
nl	Not less than (equivalent to **ge**)	**ge**	LT
ne	Not equal	—	EQ
ng	Not greater than (equivalent to **le**)	**le**	GT
so	Summary overflow	—	SO
ns	Not summary overflow	—	SO
un	Unordered (after floating-point comparison)	—	SO
nu	Not unordered (after floating-point comparison)	—	SO

Table E-17 shows the syntax for simplified branch mnemonics that incorporate CR conditions. Here, **cr**S replaces a BI operand to specify only a CR field (because the specific CR bit within the field is now part of the simplified mnemonic. Note that the default is CR0; if no **cr**S is specified, CR0 is used.

Table E-17. Branch Instructions and Simplified Mnemonics that Incorporate CR Conditions

Instruction	Standard Mnemonic	Syntax	Simplified Mnemonic	Syntax
Branch	**b** (**ba bl bla**)	target_addr	—	
Branch Conditional	**bc** (**bca bcl bcla**)	BO,BI,target_addr	**b***x* [1] (**b***x***a b***x***l b***x***la**)	**cr**S[2],target_addr
Branch Conditional to Link Register	**bclr** (**bclrl**)	BO,BI	**b***x***lr** (**b***x***lrl**)	**cr**S
Branch Conditional to Count Register	**bcctr** (**bcctrl**)	BO,BI	**b***x***ctr** (**b***x***ctrl**)	**cr**S

[1] *x* stands for one of the symbols in Table E-16, where applicable.

[2] BI can be a numeric value or an expression as shown in Table E-9.

Table E-18 shows the simplified branch mnemonics incorporating conditions.

Table E-18. Simplified Mnemonics with Comparison Conditions

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	**bca**	**bclr**	**bcctr**	**bcl**	**bcla**	**bclrl**	**bcctrl**
Branch if less than	**blt**	**blta**	**bltlr**	**bltctr**	**bltl**	**bltla**	**bltlrl**	**bltctrl**
Branch if less than or equal	**ble**	**blea**	**blelr**	**blectr**	**blel**	**blela**	**blelrl**	**blectrl**
Branch if equal	**beq**	**beqa**	**beqlr**	**beqctr**	**beql**	**beqla**	**beqlrl**	**beqctrl**
Branch if greater than or equal	**bge**	**bgea**	**bgelr**	**bgectr**	**bgel**	**bgela**	**bgelrl**	**bgectrl**
Branch if greater than	**bgt**	**bgta**	**bgtlr**	**bgtctr**	**bgtl**	**bgtla**	**bgtlrl**	**bgtctrl**
Branch if not less than	**bnl**	**bnla**	**bnllr**	**bnlctr**	**bnll**	**bnlla**	**bnllrl**	**bnlctrl**
Branch if not equal	**bne**	**bnea**	**bnelr**	**bnectr**	**bnel**	**bnela**	**bnelrl**	**bnectrl**
Branch if not greater than	**bng**	**bnga**	**bnglr**	**bngctr**	**bngl**	**bngla**	**bnglrl**	**bngctrl**

Table E-18. Simplified Mnemonics with Comparison Conditions (continued)

Branch Semantics	LR Update Not Enabled				LR Update Enabled			
	bc	bca	bclr	bcctr	bcl	bcla	bclrl	bcctrl
Branch if summary overflow	**bso**	**bsoa**	**bsolr**	**bsoctr**	**bsol**	**bsola**	**bsolrl**	**bsoctrl**
Branch if not summary overflow	**bns**	**bnsa**	**bnslr**	**bnsctr**	**bnsl**	**bnsla**	**bnslrl**	**bnsctrl**
Branch if unordered	**bun**	**buna**	**bunlr**	**bunctr**	**bunl**	**bunla**	**bunlrl**	**bunctrl**
Branch if not unordered	**bnu**	**bnua**	**bnulr**	**bnuctr**	**bnul**	**bnula**	**bnulrl**	**bnuctrl**

Instructions using the mnemonics in Table E-18 indicate the condition bit, but not the CR field. If no field is specified, CR0 is used. The CR field symbols defined in Table E-9 (**cr0**–**cr7**) are used for this operand, as shown in examples 2–4 below.

E.4.6.1 Branch Simplified Mnemonics that Incorporate CR Conditions: Examples

The following examples use the simplified mnemonics shown in Table E-18:

1. Branch if CR0 reflects not-equal condition.
   bne *target* equivalent to **bc 4,2,***target*
2. Same as (1) but condition is in CR3.
   bne cr3,*target* equivalent to **bc 4,14,***target*
3. Branch to an absolute target if CR4 specifies greater than condition, setting the LR. This is a form of conditional call.
   bgtla cr4,*target* equivalent to **bcla 12,17,***target*
4. Same as (3), but target address is in the CTR.
   bgtctrl cr4 equivalent to **bcctrl 12,17**

E.4.6.2 Branch Simplified Mnemonics that Incorporate CR Conditions: Listings

Table E-19 shows simplified branch mnemonics and syntax for **bc** and **bca** without LR updating.

Table E-19. Simplified Mnemonics for bc and bca without Comparison Conditions or LR Updating

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch if less than	**bc 12,**BI[1],target	**blt cr**S target	**bca 12,**BI[1],target	**blta cr**S target
Branch if less than or equal	**bc 4,**BI[2],target	**ble cr**S target	**bca 4,**BI[2],target	**blea cr**S target
Branch if not greater than		**bng cr**S target		**bnga cr**S target
Branch if equal	**bc 12,**BI[3],target	**beq cr**S target	**bca 12,**BI[3],target	**beqa cr**S target
Branch if greater than or equal	**bc 4,**BI[1],target	**bge cr**S target	**bca 4,**BI[1],target	**bgea cr**S target
Branch if not less than		**bnl cr**S target		**bnla cr**S target
Branch if greater than	**bc 12,**BI[2],target	**bgt cr**S target	**bca 12,**BI[2],target	**bgta cr**S target
Branch if not equal	**bc 4,**BI[3],target	**bne cr**S target	**bca 4,**BI[3],target	**bnea cr**S target

Table E-19. Simplified Mnemonics for bc and bca without Comparison Conditions or LR Updating (continued)

Branch Semantics	bc	Simplified Mnemonic	bca	Simplified Mnemonic
Branch if summary overflow	**bc 12,**BI[4],target	**bso cr**S target	**bca 12,**BI[4],target	**bsoa cr**S target
Branch if unordered		**bun cr**S target		**buna cr**S target
Branch if not summary overflow	**bc 4,**BI[4],target	**bns cr**S target	**bca 4,**BI[4],target	**bnsa cr**S target
Branch if not unordered		**bnu cr**S target		**bnua cr**S target

1 The value in the BI operand selects CR*n*[0], the LT bit.
2 The value in the BI operand selects CR*n*[1], the GT bit.
3 The value in the BI operand selects CR*n*[2], the EQ bit.
4 The value in the BI operand selects CR*n*[3], the SO bit.

Table E-20 shows simplified branch mnemonics and syntax for **bclr** and **bcctr** without LR updating.

Table E-20. Simplified Mnemonics for bclr and bcctr without Comparison Conditions and LR Updating

Branch Semantics	bclr	Simplified Mnemonic	bcctr	Simplified Mnemonic
Branch if less than	**bclr 12,**BI[1],target	**bltlr cr**S target	**bcctr 12,**BI[1],target	**bltctr cr**S target
Branch if less than or equal	**bclr 4,**BI[2],target	**blelr cr**S target	**bcctr 4,**BI[2],target	**blectr cr**S target
Branch if not greater than		**bnglr cr**S target		**bngctr cr**S target
Branch if equal	**bclr 12,**BI[3],target	**beqlr cr**S target	**bcctr 12,**BI[3],target	**beqctr cr**S target
Branch if greater than or equal	**bclr 4,**BI[1],target	**bgelr cr**S target	**bcctr 4,**BI[1],target	**bgectr cr**S target
Branch if not less than		**bnllr cr**S target		**bnlctr cr**S target
Branch if greater than	**bclr 12,**BI[2],target	**bgtlr cr**S target	**bcctr 12,**BI[2],target	**bgtctr cr**S target
Branch if not equal	**bclr 4,**BI[3],target	**bnelr cr**S target	**bcctr 4,**BI[3],target	**bnectr cr**S target
Branch if summary overflow	**bclr 12,**BI[4],target	**bsolr cr**S target	**bcctr 12,**BI[4],target	**bsoctr cr**S target
Branch if unordered		**bunlr cr**S target		**bunctr cr**S target
Branch if not summary overflow	**bclr 4,**BI[4],target	**bnslr cr**S target	**bcctr 4,**BI[4],target	**bnsctr cr**S target
Branch if not unordered		**bnulr cr**S target		**bnuctr cr**S target

1 The value in the BI operand selects CR*n*[0], the LT bit.
2 The value in the BI operand selects CR*n*[1], the GT bit.
3 The value in the BI operand selects CR*n*[2], the EQ bit.
4 The value in the BI operand selects CR*n*[3], the SO bit.

Table E-21 shows simplified branch mnemonics and syntax for **bcl** and **bcla**.

Table E-21. Simplified Mnemonics for bcl and bcla with Comparison Conditions and LR Updating

Branch Semantics	bcl	Simplified Mnemonic	bcla	Simplified Mnemonic
Branch if less than	**bcl 12,**BI[1],target	**blt cr**S target	**bcla 12,**BI[1],target	**bltla cr**S target
Branch if less than or equal	**bcl 4,**BI[2],target	**ble cr**S target	**bcla 4,**BI[2],target	**blela cr**S target
Branch if not greater than		**bng cr**S target		**bngla cr**S target
Branch if equal	**bcl 12,**BI[3],target	**beq cr**S target	**bcla 12,**BI[3],target	**beqla cr**S target
Branch if greater than or equal	**bcl 4,**BI[1],target	**bge cr**S target	**bcla 4,**BI[1],target	**bgela cr**S target
Branch if not less than		**bnl cr**S target		**bnlla cr**S target
Branch if greater than	**bcl 12,**BI[2],target	**bgt cr**S target	**bcla 12,**BI[2],target	**bgtla cr**S target
Branch if not equal	**bcl 4,**BI[3],target	**bne cr**S target	**bcla 4,**BI[3],target	**bnela cr**S target
Branch if summary overflow	**bcl 12,**BI[4],target	**bso cr**S target	**bcla 12,**BI[4],target	**bsola cr**S target
Branch if unordered		**bun cr**S target		**bunla cr**S target
Branch if not summary overflow	**bcl 4,**BI[4],target	**bns cr**S target	**bcla 4,**BI[4],target	**bnsla cr**S target
Branch if not unordered		**bnu cr**S target		**bnula cr**S target

1 The value in the BI operand selects CR*n*[0], the LT bit.
2 The value in the BI operand selects CR*n*[1], the GT bit.
3 The value in the BI operand selects CR*n*[2], the EQ bit.
4 The value in the BI operand selects CR*n*[3], the SO bit.

Table E-22 shows the simplified branch mnemonics and syntax for **bclrl** and **bcctrl** with LR updating.

Table E-22. Simplified Mnemonics for bclrl and bcctrl with Comparison Conditions and LR Updating

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch if less than	**bclrl 12,**BI[1],target	**bltlrl cr**S target	**bcctrl 12,**BI[1],target	**bltctrl cr**S target
Branch if less than or equal	**bclrl 4,**BI[2],target	**blelrl cr**S target	**bcctrl 4,**BI[2],target	**blectrl cr**S target
Branch if not greater than		**bnglrl cr**S target		**bngctrl cr**S target
Branch if equal	**bclrl 12,**BI[3],target	**beqlrl cr**S target	**bcctrl 12,**BI[3],target	**beqctrl cr**S target
Branch if greater than or equal	**bclrl 4,**BI[1],target	**bgelrl cr**S target	**bcctrl 4,**BI[1],target	**bgectrl cr**S target
Branch if not less than		**bnllrl cr**S target		**bnlctrl cr**S target
Branch if greater than	**bclrl 12,**BI[2],target	**bgtlrl cr**S target	**bcctrl 12,**BI[2],target	**bgtctrl cr**S target
Branch if not equal	**bclrl 4,**BI[3],target	**bnelrl cr**S target	**bcctrl 4,**BI[3],target	**bnectrl cr**S target
Branch if summary overflow	**bclrl 12,**BI[4],target	**bsolrl cr**S target	**bcctrl 12,**BI[4],target	**bsoctrl cr**S target
Branch if unordered		**bunlrl cr**S target		**bunctrl cr**S target

Table E-22. Simplified Mnemonics for bclrl and bcctrl with Comparison Conditions and LR Updating (continued)

Branch Semantics	bclrl	Simplified Mnemonic	bcctrl	Simplified Mnemonic
Branch if not summary overflow	**bclrl 4,**BI[4]**,**target	**bnslrl cr**S target	**bcctrl 4,**BI[4]**,**target	**bnsctrl cr**S target
Branch if not unordered		**bnulrl cr**S target		**bnuctrl cr**S target

1 The value in the BI operand selects CR*n*[0], the LT bit.
2 The value in the BI operand selects CR*n*[1], the GT bit.
3 The value in the BI operand selects CR*n*[2], the EQ bit.
4 The value in the BI operand selects CR*n*[3], the SO bit.

E.5 Compare Word Simplified Mnemonics

In compare word instructions, the L operand indicates a word (L = 0) or double-word (L = 1). Simplified mnemonics in Table E-23 eliminate the L operand for word comparisons.

Table E-23. Word Compare Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to:
Compare Word Immediate	**cmpwi cr**D,**r**A,SIMM	**cmpi cr**D,**0,r**A,SIMM
Compare Word	**cmpw cr**D,**r**A,**r**B	**cmp cr**D,**0,r**A,**r**B
Compare Logical Word Immediate	**cmplwi cr**D,**r**A,UIMM	**cmpli cr**D,**0,r**A,UIMM
Compare Logical Word	**cmplw cr**D,**r**A,**r**B	**cmpl cr**D,**0,r**A,**r**B

As with branch mnemonics, the **cr**D field of a compare instruction can be omitted if CR0 is used, as shown in examples 1 and 3 below. Otherwise, the target CR field must be specified as the first operand. The following examples use word compare mnemonics:

1. Compare **r**A with immediate value 100 as signed 32-bit integers and place result in CR0.
   cmpwi rA**,100** equivalent to **cmpi 0,0,r**A**,100**
2. Same as (1), but place results in CR4.
   cmpwi cr4,rA**,100** equivalent to **cmpi 4,0,r**A**,100**
3. Compare **r**A and **r**B as unsigned 32-bit integers and place result in CR0.
   cmplw rA**,r**B equivalent to **cmpl 0,0,r**A**,r**B

E.6 Condition Register Logical Simplified Mnemonics

The CR logical instructions, shown in Table E-24, can be used to set, clear, copy, or invert a given CR bit. Simplified mnemonics allow these operations to be coded easily. Note that the symbols defined in Table E-8 can be used to identify the CR bit.

Table E-24. Condition Register Logical Simplified Mnemonics

Operation	Simplified Mnemonic	Equivalent to
Condition register set	**crset** b*x*	**creqv** b*x*,b*x*,b*x*
Condition register clear	**crclr** b*x*	**crxor** b*x*,b*x*,b*x*
Condition register move	**crmove** b*x*,b*y*	**cror** b*x*,b*y*,b*y*
Condition register not	**crnot** b*x*,b*y*	**crnor** b*x*,b*y*,b*y*

Examples using the CR logical mnemonics follow:

1. Set CR[57].
   crset 25 equivalent to **creqv 25,25,25**
2. Clear CR0[SO].
   crclr so equivalent to **crxor 3,3,3**
3. Same as (2), but clear CR3[SO].
   crclr 4 * cr3 + so equivalent to **crxor 15,15,15**
4. Invert the CR0[EQ].
   crnot eq,eq equivalent to **crnor 2,2,2**
5. Same as (4), but CR4[EQ] is inverted and the result is placed into CR5[EQ].
   crnot 4 * cr5 + eq, 4 * cr4 + eq equivalent to **crnor 22,18,18**

E.7 Trap Instructions Simplified Mnemonics

The codes in Table E-25 have been adopted for the most common combinations of trap conditions.

Table E-25. Standard Codes for Trap Instructions

Code	Description	TO Encoding	<	>	=	<U[1]	>U [2]
lt	Less than	16	1	0	0	0	0
le	Less than or equal	20	1	0	1	0	0
eq	Equal	4	0	0	1	0	0
ge	Greater than or equal	12	0	1	1	0	0
gt	Greater than	8	0	1	0	0	0
nl	Not less than	12	0	1	1	0	0
ne	Not equal	24	1	1	0	0	0
ng	Not greater than	20	1	0	1	0	0
llt	Logically less than	2	0	0	0	1	0

Table E-25. Standard Codes for Trap Instructions (continued)

Code	Description	TO Encoding	<	>	=	<U[1]	>U [2]
lle	Logically less than or equal	6	0	0	1	1	0
lge	Logically greater than or equal	5	0	0	1	0	1
lgt	Logically greater than	1	0	0	0	0	1
lnl	Logically not less than	5	0	0	1	0	1
lng	Logically not greater than	6	0	0	1	1	0
—	Unconditional	31	1	1	1	1	1

[1] The symbol '<U' indicates an unsigned less-than evaluation is performed.

[2] The symbol '>U' indicates an unsigned greater-than evaluation is performed.

The mnemonics in Table E-26 are variations of trap instructions, with the most useful TO values represented in the mnemonic rather than specified as a numeric operand.

Table E-26. Trap Simplified Mnemonics

Trap Semantics	32-Bit Comparison	
	twi Immediate	tw Register
Trap unconditionally	—	**trap**
Trap if less than	**twlti**	**twlt**
Trap if less than or equal	**twlei**	**twle**
Trap if equal	**tweqi**	**tweq**
Trap if greater than or equal	**twgei**	**twge**
Trap if greater than	**twgti**	**twgt**
Trap if not less than	**twnli**	**twnl**
Trap if not equal	**twnei**	**twne**
Trap if not greater than	**twngi**	**twng**
Trap if logically less than	**twllti**	**twllt**
Trap if logically less than or equal	**twllei**	**twlle**
Trap if logically greater than or equal	**twlgei**	**twlge**
Trap if logically greater than	**twlgti**	**twlgt**
Trap if logically not less than	**twlnli**	**twlnl**
Trap if logically not greater than	**twlngi**	**twlng**

The following examples use the trap mnemonics shown in Table E-26:

1. Trap if **r**A is not zero.
   twnei rA**,0** equivalent to **twi 24,r**A**,0**
2. Trap if **r**A is not equal to **r**B.
   twne rA**, r**B equivalent to **tw 24,r**A**,r**B
3. Trap if **r**A is logically greater than 0x7FF.
   twlgti rA**,** 0x7FF equivalent to **twi 1,r**A**,** 0x7FF
4. Trap unconditionally.
   trap equivalent to **tw 31,0,0**

Trap instructions evaluate a trap condition as follows: The contents of **r**A are compared with either the sign-extended SIMM field or the contents of **r**B, depending on the trap instruction.

The comparison results in five conditions that are ANDed with operand TO. If the result is not 0, the trap exception handler is invoked. See Table E-27 for these conditions.

Table E-27. TO Operand Bit Encoding

TO Bit	ANDed with Condition
0	Less than, using signed comparison
1	Greater than, using signed comparison
2	Equal
3	Less than, using unsigned comparison
4	Greater than, using unsigned comparison

E.8 Simplified Mnemonics for Accessing SPRs

The **mtspr** and **mfspr** instructions specify a special-purpose register (SPR) as a numeric operand. Simplified mnemonics are provided that represent the SPR in the mnemonic rather than requiring it to be coded as a numeric operand. The pattern for **mtspr** and **mfspr** simplified mnemonics is straightforward: replace the -**spr** portion of the mnemonic with the abbreviation for the spr (for example XER, SRR0, or LR), eliminate the SPRN operand, leaving the source or destination GPR operand, **r**S or **r**D.

Following are examples using the SPR simplified mnemonics:

1. Copy the contents of **r**S to the XER.
   mtxer rS equivalent to **mtspr 1,r**S
2. Copy the contents of the LR to **r**S.
   mflr rD equivalent to **mfspr r**D**,8**
3. Copy the contents of **r**S to the CTR.
   mtctr rS equivalent to **mtspr 9,r**S

There is an additional simplified mnemonic formula for accessing IBATs, DBATs, and SPRGs, although not all of these more complicated simplified mnemonics are supported by all assemblers. These are shown in Table E-28 along with the equivalent simplified mnemonic using the formula described above.

Table E-28. Additional Simplified Mnemonics for Accessing IBATs, DBATs, and SPRGs

SPR	Move to SPR		Move from SPR	
	Simplified Mnemonic	**Equivalent to**	**Simplified Mnemonic**	**Equivalent to**
DBAT register, lower	**mtdbatl** *n,* **r**S	**mtspr** 537 + (2 * *n*)**,r**S	**mfdbatl r**D, *n*	**mfspr r**D,537 + (2 * *n*)
	mtdbatl*n,* **r**S		**mfdbatl***n* **r**D	
DBAT register, lower	**mtdbatu** *n,* **r**S	**mtspr** 536 + (2 **n*)**,r**S	**mfdbatu r**D, *n*	**mfspr r**D,536 + (2 **n*)
	mtdbatu*n,* **r**S		**mfdbatu***n* **r**D	
IBAT register, lower	**mtibatl** *n,* **r**S	**mtspr** 529 + (2 * *n*)**,r**S	**mfibatl r**D, *n*	**mfspr r**D,529 + (2 * *n*)
	mtibatl*n,* **r**S		**mfibatl***n* **r**D	
IBAT register, upper	**mtibatu** *n,* **r**S	**mtspr** 528 + (2 * *n*)**,r**S	**mfibatu r**D, *n*	**mfspr r**D,528 + (2 * *n*)
	mtibatu*n,* **r**S		**mfibatu***n* **r**D	
SPRGs	**mtsprg** *n,* **r**S	**mtspr** 272 + *n***,r**S	**mfsprg r**D, *n*	**mfspr r**D,272 + *n*
	mtsprg*n,* **r**S		**mfsprg***n* **r**D	

E.9 Recommended Simplified Mnemonics

This section describes commonly-used operations (such as no-op, load immediate, load address, move register, and complement register).

E.9.1 No-Op (nop)

Many instructions can be coded in a way that, effectively, no operation is performed. An additional mnemonic is provided for the preferred form of no-op. If an implementation performs any type of run-time optimization related to no-ops, the preferred form is the following:

nop equivalent to**ori 0,0,0**

E.9.2 Load Immediate (li)

The **addi** and **addis** instructions can be used to load an immediate value into a register. Additional mnemonics are provided to convey the idea that no addition is being performed but that data is being moved from the immediate operand of the instruction to a register.

1. Load a 16-bit signed immediate value into **r**D.
   li rD**,**value equivalent to **addi r**D**,0,**value
2. Load a 16-bit signed immediate value, shifted left by 16 bits, into **r**D.
   lis rD**,**value equivalent to **addis r**D**,0,**value

E.9.3 Load Address (la)

This mnemonic permits computing the value of a base-displacement operand, using the **addi** instruction that normally requires a separate register and immediate operands.

la **r**D**,**d(**r**A) equivalent to **addi** **r**D**,r**A**,**d

The **la** mnemonic is useful for obtaining the address of a variable specified by name, allowing the assembler to supply the base register number and compute the displacement. If the variable *v* is located at offset d*v* bytes from the address in **r***v*, and the assembler has been told to use **r***v* as a base for references to the data structure containing *v*, the following line causes the address of *v* to be loaded into **r**D:

la **r**D**,***v* equivalent to **addi** **r**D**,r***v***,**d*v*

E.9.4 Move Register (mr)

Several instructions can be coded to copy the contents of one register to another. A simplified mnemonic is provided that signifies that no computation is being performed, but merely that data is being moved from one register to another.

The following instruction copies the contents of **r**S into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

mr **r**A**,r**S equivalent to **or** **r**A**,r**S**,r**S

E.9.5 Complement Register (not)

Several instructions can be coded in a way that they complement the contents of one register and place the result into another register. A simplified mnemonic is provided that allows this operation to be coded easily.

The following instruction complements the contents of **r**S and places the result into **r**A. This mnemonic can be coded with a dot (**.**) suffix to cause the Rc bit to be set in the underlying instruction.

not **r**A**,r**S equivalent to **nor** **r**A**,r**S**,r**S

E.9.6 Move to Condition Register (mtcr)

This mnemonic permits copying the contents of a GPR to the CR, using the same syntax as the **mfcr** instruction.

mtcr **r**S equivalent to **mtcrf** 0xFF**,r**S

E.10 Comprehensive List of Simplified Mnemonics

Table E-29 lists simplified mnemonics. Note that compiler designers may implement additional simplified mnemonics not listed here.

Table E-29. Simplified Mnemonics

Simplified Mnemonic	Mnemonic	Instruction
bctr [1]	**bcctr 20,0**	Branch unconditionally (**bcctr** without LR update)
bctrl [1]	**bcctrl 20,0**	Branch unconditionally (**bcctrl** with LR Update)
bdnz target [1]	**bc 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bc** without LR update)
bdnza target [1]	**bca 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bca** without LR update)
bdnzf BI,target	**bc 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bc** without LR update)
bdnzfa BI,target	**bca 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bca** without LR update)
bdnzfl BI,target	**bcl 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bcl** with LR update)
bdnzfla BI,target	**bcla 0,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition false (**bcla** with LR update)
bdnzflr BI	**bclr 0,BI**	Decrement CTR, branch if CTR ≠ 0 and condition false (**bclr** without LR update)
bdnzflrl BI	**bclrl 0,BI**	Decrement CTR, branch if CTR ≠ 0 and condition false (**bclrl** with LR Update)
bdnzl target [1]	**bcl 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bcl** with LR update)
bdnzla target [1]	**bcla 16,0,**target	Decrement CTR, branch if CTR ≠ 0 (**bcla** with LR update)
bdnzlr BI	**bclr 16,BI**	Decrement CTR, branch if CTR ≠ 0 (**bclr** without LR update)
bdnzlrl [1]	**bclrl 16,0**	Decrement CTR, branch if CTR ≠ 0 (**bclrl** with LR Update)
bdnzt BI,target	**bc 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bc** without LR update)
bdnzta BI,target	**bca 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bc**a without LR update)
bdnztl BI,target	**bcl 8,0,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bcl** with LR update)
bdnztla BI,target	**bcla 8,BI,**target	Decrement CTR, branch if CTR ≠ 0 and condition true (**bcla** with LR update)
bdnztlr BI	**bclr 8,BI**	Decrement CTR, branch if CTR ≠ 0 and condition true (**bclr** without LR update)
bdnztlr BI	**bclr 8,BI**	Decrement CTR, branch if CTR = 0 and condition true (**bclr** without LR update)
bdnztlrl BI	**bclrl 8,BI**	Decrement CTR, branch if CTR ≠ 0 and condition true (**bclrl** with LR Update)
bdz target [1]	**bc 18,0,**target	Decrement CTR, branch if CTR = 0 (**bc** without LR update)
bdza target [1]	**bca 18,0,**target	Decrement CTR, branch if CTR = 0 (**bca** without LR update)

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bdzf BI,target	**bc 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bc** without LR update)
bdzfa BI,target	**bca 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bca** without LR update)
bdzfl BI,target	**bcl 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bcl** with LR update)
bdzfla BI,target	**bcla 2,BI,**target	Decrement CTR, branch if CTR = 0 and condition false (**bcla** with LR update)
bdzflr BI	**bclr 2,BI**	Decrement CTR, branch if CTR = 0 and condition false (**bclr** without LR update)
bdzflrl BI	**bclrl 2,BI**	Decrement CTR, branch if CTR = 0 and condition false (**bclrl** with LR Update)
bdzl target [1]	**bcl 18,BI,**target	Decrement CTR, branch if CTR = 0 (**bcl** with LR update)
bdzla target [1]	**bcla 18,BI,**target	Decrement CTR, branch if CTR = 0 (**bcla** with LR update)
bdzlr [1]	**bclr 18,0**	Decrement CTR, branch if CTR = 0 (**bclr** without LR update)
bdzlrl [1]	**bclrl 18,0**	Decrement CTR, branch if CTR = 0 (**bclrl** with LR Update)
bdzt BI,target	**bc 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bc** without LR update)
bdzta BI,target	**bca 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bca** without LR update)
bdztl BI,target	**bcl 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bcl** with LR update)
bdztla BI,target	**bcla 10,BI,**target	Decrement CTR, branch if CTR = 0 and condition true (**bcla** with LR update)
bdztlrl BI	**bclrl 10, BI**	Decrement CTR, branch if CTR = 0 and condition true (**bclrl** with LR Update)
beq crS target	**bc 12,**BI[2]**,**target	Branch if equal (**bc** without comparison conditions or LR updating)
beqa crS target	**bca 12,**BI[2]**,**target	Branch if equal (**bca** without comparison conditions or LR updating)
beqctr crS target	**bcctr 12,**BI[2]**,**target	Branch if equal (**bcctr** without comparison conditions and LR updating)
beqctrl crS target	**bcctrl 12,**BI[2]**,**target	Branch if equal (**bcctrl** with comparison conditions and LR update)
beql crS target	**bcl 12,**BI[2]**,**target	Branch if equal (**bcl** with comparison conditions and LR updating)
beqla crS target	**bcla 12,**BI[2]**,**target	Branch if equal (**bcla** with comparison conditions and LR updating)
beqlr crS target	**bclr 12,**BI[2]**,**target	Branch if equal (**bclr** without comparison conditions and LR updating)

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
beqlrl crS target	**bclrl 12,**BI[2],target	Branch if equal (**bclrl** with comparison conditions and LR update)
bf BI,target	**bc 4,BI,**target	Branch if condition false [3] (**bc** without LR update)
bfa BI,target	**bca 4,BI,**target	Branch if condition false [3] (**bca** without LR update)
bfctr BI	**bcctr 4,BI**	Branch if condition false [3] (**bcctr** without LR update)
bfctrl BI	**bcctrl 4,BI**	Branch if condition false [3](**bcctrl** with LR Update)
bfl BI,target	**bcl 4,BI,**target	Branch if condition false [3] (**bcl** with LR update)
bfla BI,target	**bcla 4,BI,**target	Branch if condition false [3] (**bcla** with LR update)
bflr BI	**bclr 4,BI**	Branch if condition false [3] (**bclr** without LR update)
bflrl BI	**bclrl 4,BI**	Branch if condition false [3](**bclrl** with LR Update)
bge crS target	**bc 4,**BI[4],target	Branch if greater than or equal (**bc** without comparison conditions or LR updating)
bgea crS target	**bca 4,**BI[4],target	Branch if greater than or equal (**bca** without comparison conditions or LR updating)
bgectr crS target	**bcctr 4,**BI[4],target	Branch if greater than or equal (**bcctr** without comparison conditions and LR updating)
bgectrl crS target	**bcctrl 4,**BI[4],target	Branch if greater than or equal (**bcctrl** with comparison conditions and LR update)
bgel crS target	**bcl 4,**BI[4],target	Branch if greater than or equal (**bcl** with comparison conditions and LR updating)
bgela crS target	**bcla 4,**BI[4],target	Branch if greater than or equal (**bcla** with comparison conditions and LR updating)
bgelr crS target	**bclr 4,**BI[4],target	Branch if greater than or equal (**bclr** without comparison conditions and LR updating)
bgelrl crS target	**bclrl 4,**BI[4],target	Branch if greater than or equal (**bclrl** with comparison conditions and LR update)
bgt crS target	**bc 12,**BI[5],target	Branch if greater than (**bc** without comparison conditions or LR updating)
bgta crS target	**bca 12,**BI[5],target	Branch if greater than (**bca** without comparison conditions or LR updating)
bgtctr crS target	**bcctr 12,**BI[5],target	Branch if greater than (**bcctr** without comparison conditions and LR updating)
bgtctrl crS target	**bcctrl 12,**BI[5],target	Branch if greater than (**bcctrl** with comparison conditions and LR update)
bgtl crS target	**bcl 12,**BI[5],target	Branch if greater than (**bcl** with comparison conditions and LR updating)
bgtla crS target	**bcla 12,**BI[5],target	Branch if greater than (**bcla** with comparison conditions and LR updating)
bgtlr crS target	**bclr 12,**BI[5],target	Branch if greater than (**bclr** without comparison conditions and LR updating)
bgtlrl crS target	**bclrl 12,**BI[5],target	Branch if greater than (**bclrl** with comparison conditions and LR update)

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
ble crS target	**bc 4,**BI[5],target	Branch if less than or equal (**bc** without comparison conditions or LR updating)
blea crS target	**bca 4,**BI[5],target	Branch if less than or equal (**bca** without comparison conditions or LR updating)
blectr crS target	**bcctr 4,**BI[5],target	Branch if less than or equal (**bcctr** without comparison conditions and LR updating)
blectrl crS target	**bcctrl 4,**BI[5],target	Branch if less than or equal (**bcctrl** with comparison conditions and LR update)
blel crS target	**bcl 4,**BI[5],target	Branch if less than or equal (**bcl** with comparison conditions and LR updating)
blela crS target	**bcla 4,**BI[5],target	Branch if less than or equal (**bcla** with comparison conditions and LR updating)
blelr crS target	**bclr 4,**BI[5],target	Branch if less than or equal (**bclr** without comparison conditions and LR updating)
blelrl crS target	**bclrl 4,**BI[5],target	Branch if less than or equal (**bclrl** with comparison conditions and LR update)
blr [1]	**bclr 20,0**	Branch unconditionally (**bclr** without LR update)
blrl [1]	**bclrl 20,0**	Branch unconditionally (**bclrl** with LR Update)
blt crS target	**bc 12,**BI,target	Branch if less than (**bc** without comparison conditions or LR updating)
blta crS target	**bca 12,**BI[4],target	Branch if less than (**bca** without comparison conditions or LR updating)
bltctr crS target	**bcctr 12,**BI[4],target	Branch if less than (**bcctr** without comparison conditions and LR updating)
bltctrl crS target	**bcctrl 12,**BI[4],target	Branch if less than (**bcctrl** with comparison conditions and LR update)
bltl crS target	**bcl 12,**BI[4],target	Branch if less than (**bcl** with comparison conditions and LR updating)
bltla crS target	**bcla 12,**BI[4],target	Branch if less than (**bcla** with comparison conditions and LR updating)
bltlr crS target	**bclr 12,**BI[4],target	Branch if less than (**bclr** without comparison conditions and LR updating)
bltlrl crS target	**bclrl 12,**BI[4],target	Branch if less than (**bclrl** with comparison conditions and LR update)
bne crS target	**bc 4,**BI[3],target	Branch if not equal (**bc** without comparison conditions or LR updating)
bnea crS target	**bca 4,**BI[3],target	Branch if not equal (**bca** without comparison conditions or LR updating)
bnectr crS target	**bcctr 4,**BI[3],target	Branch if not equal (**bcctr** without comparison conditions and LR updating)
bnectrl crS target	**bcctrl 4,**BI[3],target	Branch if not equal (**bcctrl** with comparison conditions and LR update)
bnel crS target	**bcl 4,**BI[3],target	Branch if not equal (**bcl** with comparison conditions and LR updating)

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bnela crS target	**bcla 4,**BI[3],target	Branch if not equal (**bcla** with comparison conditions and LR updating)
bnelr crS target	**bclr 4,**BI[3],target	Branch if not equal (**bclr** without comparison conditions and LR updating)
bnelrl crS target	**bclrl 4,**BI[3],target	Branch if not equal (**bclrl** with comparison conditions and LR update)
bng crS target	**bc 4,**BI[5],target	Branch if not greater than (**bc** without comparison conditions or LR updating)
bnga crS target	**bca 4,**BI[5],target	Branch if not greater than (**bca** without comparison conditions or LR updating)
bngctr crS target	**bcctr 4,**BI[5],target	Branch if not greater than (**bcctr** without comparison conditions and LR updating)
bngctrl crS target	**bcctrl 4,**BI[5],target	Branch if not greater than (**bcctrl** with comparison conditions and LR update)
bngl crS target	**bcl 4,**BI[5],target	Branch if not greater than (**bcl** with comparison conditions and LR updating)
bngla crS target	**bcla 4,**BI[5],target	Branch if not greater than (**bcla** with comparison conditions and LR updating)
bnglr crS target	**bclr 4,**BI[5],target	Branch if not greater than (**bclr** without comparison conditions and LR updating)
bnglrl crS target	**bclrl 4,**BI[5],target	Branch if not greater than (**bclrl** with comparison conditions and LR update)
bnl crS target	**bc 4,**BI[4],target	Branch if not less than (**bc** without comparison conditions or LR updating)
bnla crS target	**bca 4,**BI[4],target	Branch if not less than (**bca** without comparison conditions or LR updating)
bnlctr crS target	**bcctr 4,**BI[4],target	Branch if not less than (**bcctr** without comparison conditions and LR updating)
bnlctrl crS target	**bcctrl 4,**BI[4],target	Branch if not less than (**bcctrl** with comparison conditions and LR update)
bnll crS target	**bcl 4,**BI[4],target	Branch if not less than (**bcl** with comparison conditions and LR updating)
bnlla crS target	**bcla 4,**BI[4],target	Branch if not less than (**bcla** with comparison conditions and LR updating)
bnllr crS target	**bclr 4,**BI[4],target	Branch if not less than (**bclr** without comparison conditions and LR updating)
bnllrl crS target	**bclrl 4,**BI[4],target	Branch if not less than (**bclrl** with comparison conditions and LR update)
bns crS target	**bc 4,**BI[6],target	Branch if not summary overflow (**bc** without comparison conditions or LR updating)
bnsa crS target	**bca 4,**BI[6],target	Branch if not summary overflow (**bca** without comparison conditions or LR updating)
bnsctr crS target	**bcctr 4,**BI[6],target	Branch if not summary overflow (**bcctr** without comparison conditions and LR updating)

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
bnsctrl crS target	**bcctrl 4,**BI[6]**,**target	Branch if not summary overflow (**bcctrl** with comparison conditions and LR update)
bnsl crS target	**bcl 4,**BI[6]**,**target	Branch if not summary overflow (**bcl** with comparison conditions and LR updating)
bnsla crS target	**bcla 4,**BI[6]**,**target	Branch if not summary overflow (**bcla** with comparison conditions and LR updating)
bnslr crS target	**bclr 4,**BI[6]**,**target	Branch if not summary overflow (**bclr** without comparison conditions and LR updating)
bnslrl crS target	**bclrl 4,**BI[6]**,**target	Branch if not summary overflow (**bclrl** with comparison conditions and LR update)
bnu crS target	**bc 4,**BI[6]**,**target	Branch if not unordered (**bc** without comparison conditions or LR updating)
bnua crS target	**bca 4,**BI[6]**,**target	Branch if not unordered (**bca** without comparison conditions or LR updating)
bnuctr crS target	**bcctr 4,**BI[6]**,**target	Branch if not unordered (**bcctr** without comparison conditions and LR updating)
bnuctrl crS target	**bcctrl 4,**BI[6]**,**target	Branch if not unordered (**bcctrl** with comparison conditions and LR update)
bnul crS target	**bcl 4,**BI[6]**,**target	Branch if not unordered (**bcl** with comparison conditions and LR updating)
bnula crS target	**bcla 4,**BI[6]**,**target	Branch if not unordered (**bcla** with comparison conditions and LR updating)
bnulr crS target	**bclr 4,**BI[6]**,**target	Branch if not unordered (**bclr** without comparison conditions and LR updating)
bnulrl crS target	**bclrl 4,**BI[6]**,**target	Branch if not unordered (**bclrl** with comparison conditions and LR update)
bso crS target	**bc 12,**BI[6]**,**target	Branch if summary overflow (**bc** without comparison conditions or LR updating)
bsoa crS target	**bca 12,**BI[6]**,**target	Branch if summary overflow (**bca** without comparison conditions or LR updating)
bsoctr crS target	**bcctr 12,**BI[6]**,**target	Branch if summary overflow (**bcctr** without comparison conditions and LR updating)
bsoctrl crS target	**bcctrl 12,**BI[6]**,**target	Branch if summary overflow (**bcctrl** with comparison conditions and LR update)
bsol crS target	**bcl 12,**BI[6]**,**target	Branch if summary overflow (**bcl** with comparison conditions and LR updating)
bsola crS target	**bcla 12,**BI[6]**,**target	Branch if summary overflow (**bcla** with comparison conditions and LR updating)
bsolr crS target	**bclr 12,**BI[6]**,**target	Branch if summary overflow (**bclr** without comparison conditions and LR updating)
bsolrl crS target	**bclrl 12,**BI[6]**,**target	Branch if summary overflow (**bclrl** with comparison conditions and LR update)
bt BI,target	**bc 12,BI,**target	Branch if condition true[3] (**bc** without LR update)
bta BI,target	**bca 12,BI,**target	Branch if condition true[3] (**bca** without LR update)

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
btctr BI	**bcctr 12,BI**	Branch if condition true [3] (**bcctr** without LR update)
btctrl BI	**bcctrl 12,BI**	Branch if condition true [3] (**bcctrl** with LR Update)
btl BI,target	**bcl 12,BI**,target	Branch if condition true [3] (**bcl** with LR update)
btla BI,target	**bcla 12,BI**,target	Branch if condition true [3] (**bcla** with LR update)
btlr BI	**bclr 12,BI**	Branch if condition true [3] (**bclr** without LR update)
btlrl BI	**bclrl 12,BI**	Branch if condition true [3] (**bclrl** with LR Update)
bun crS target	**bc 12,**BI[6]**,**target	Branch if unordered (**bc** without comparison conditions or LR updating)
buna crS target	**bca 12,**BI[6]**,**target	Branch if unordered (**bca** without comparison conditions or LR updating)
bunctr crS target	**bcctr 12,**BI[6]**,**target	Branch if unordered (**bcctr** without comparison conditions and LR updating)
bunctrl crS target	**bcctrl 12,**BI[6]**,**target	Branch if unordered (**bcctrl** with comparison conditions and LR update)
bunl crS target	**bcl 12,**BI[6]**,**target	Branch if unordered (**bcl** with comparison conditions and LR updating)
bunla crS target	**bcla 12,**BI[6]**,**target	Branch if unordered (**bcla** with comparison conditions and LR updating)
bunlr crS target	**bclr 12,**BI[6]**,**target	Branch if unordered (**bclr** without comparison conditions and LR updating)
bunlrl crS target	**bclrl 12,**BI[6]**,**target	Branch if unordered (**bclrl** with comparison conditions and LR update)
clrlslwi rA,**r**S,*b*,*n* ($n \le b \le 31$)	**rlwinm r**A,**r**S,*n*,*b* – *n*,31 – *n*	Clear left and shift left word immediate
clrlwi rA,**r**S,*n* ($n < 32$)	**rlwinm r**A,**r**S,**0**,*n*,**31**	Clear left word immediate
clrrwi rA,**r**S,*n* ($n < 32$)	**rlwinm r**A,**r**S,**0,0**,31 – *n*	Clear right word immediate
cmplw crD,**r**A,**r**B	**cmpl cr**D,**0,r**A,**r**B	Compare logical word
cmplwi crD,**r**A,UIMM	**cmpli cr**D,**0,r**A,UIMM	Compare logical word immediate
cmpw crD,**r**A,**r**B	**cmp cr**D,**0,r**A,**r**B	Compare word
cmpwi crD,**r**A,SIMM	**cmpi cr**D,**0,r**A,SIMM	Compare word immediate
crclr bx	**crxor** bx,bx,bx	Condition register clear
crmove bx,by	**cror** bx,by,by	Condition register move
crnot bx,by	**crnor** bx,by,by	Condition register not
crset bx	**creqv** bx,bx,bx	Condition register set
evsubiw rD,**r**B,UIMM	**evsubifw r**D,UIMM,**r**B	Vector subtract word immediate
evsubw rD,**r**B,**r**A	**evsubfw r**D,**r**A,**r**B	Vector subtract word
extlwi rA,**r**S,*n*,*b* ($n > 0$)	**rlwinm r**A,**r**S,*b*,**0**,*n* – 1	Extract and left justify word immediate
extrwi rA,**r**S,*n*,*b* ($n > 0$)	**rlwinm r**A,**r**S,*b* + *n*, 32 – *n*,**31**	Extract and right justify word immediate
inslwi rA,**r**S,*n*,*b* ($n > 0$)	**rlwimi r**A,**r**S,32 – *b*,*b*,(*b* + *n*) – 1	Insert from left word immediate
insrwi rA,**r**S,*n*,*b* ($n > 0$)	**rlwimi r**A,**r**S,32 – (*b* + *n*),*b*,(*b* + *n*) – 1	Insert from right word immediate

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
la rD,d(rA)	**addi** rD,rA,d	Load address
li rD,value	**addi** rD,**0**,value	Load immediate
lis rD,value	**addis** rD,**0**,value	Load immediate signed
mf*spr* rD	**mfspr** rD,SPRN	Move from SPR (see Section E.8, “Simplified Mnemonics for Accessing SPRs.”)
mr rA,rS	**or** rA,rS,rS	Move register
mtcr rS	**mtcrf** 0xFF,rS	Move to Condition Register
mt*spr* rS	**mfspr** SPRN,rS	Move to SPR (see Section E.8, “Simplified Mnemonics for Accessing SPRs.”)
nop	**ori 0,0,0**	No-op
not rA,rS	**nor** rA,rS,rS	NOT
not rA,rS	**nor** rA,rS,rS	Complement register
rotlw rA,rS,rB	**rlwnm** rA,rS,rB,**0,31**	Rotate left word
rotlwi rA,rS,*n*	**rlwinm** rA,rS,*n*,**0,31**	Rotate left word immediate
rotrwi rA,rS,*n*	**rlwinm** rA,rS,32 – *n*,**0,31**	Rotate right word immediate
slwi rA,rS,*n* (*n* < 32)	**rlwinm** rA,rS,*n*,**0**,31 – *n*	Shift left word immediate
srwi rA,rS,*n* (*n* < 32)	**rlwinm** rA,rS,32 – *n*,*n*,**31**	Shift right word immediate
sub rD,rA,rB	**subf** rD,rB,rA	Subtract from
subc rD,rA,rB	**subfc** rD,rB,rA	Subtract from carrying
subi rD,rA,value	**addi** rD,rA,–value	Subtract immediate
subic rD,rA,value	**addic** rD,rA,–value	Subtract immediate carrying
subic. rD,rA,value	**addic.** rD,rA,–value	Subtract immediate carrying
subis rD,rA,value	**addis** rD,rA,–value	Subtract immediate signed
tweq rA,SIMM	**tw 4,**rA,SIMM	Trap if equal
tweqi rA,SIMM	**twi 4,**rA,SIMM	Trap immediate if equal
twge rA,SIMM	**tw 12,**rA,SIMM	Trap if greater than or equal
twgei rA,SIMM	**twi 12,**rA,SIMM	Trap immediate if greater than or equal
twgt rA,SIMM	**tw 8,**rA,SIMM	Trap if greater than
twgti rA,SIMM	**twi 8,**rA,SIMM	Trap immediate if greater than
twle rA,SIMM	**tw 20,**rA,SIMM	Trap if less than or equal
twlei rA,SIMM	**twi 20,**rA,SIMM	Trap immediate if less than or equal
twlge rA,SIMM	**tw 12,**rA,SIMM	Trap if logically greater than or equal
twlgei rA,SIMM	**twi 12,**rA,SIMM	Trap immediate if logically greater than or equal
twlgt rA,SIMM	**tw 1,**rA,SIMM	Trap if logically greater than
twlgti rA,SIMM	**twi 1,**rA,SIMM	Trap immediate if logically greater than
twlle rA,SIMM	**tw 6,**rA,SIMM	Trap if logically less than or equal
twllei rA,SIMM	**twi 6,**rA,SIMM	Trap immediate if logically less than or equal
twllt rA,SIMM	**tw 2,**rA,SIMM	Trap if logically less than

Table E-29. Simplified Mnemonics (continued)

Simplified Mnemonic	Mnemonic	Instruction
twllti rA,SIMM	**twi 2,**rA,SIMM	Trap immediate if logically less than
twlng rA,SIMM	**tw 6,**rA,SIMM	Trap if logically not greater than
twlngi rA,SIMM	**twi 6,**rA,SIMM	Trap immediate if logically not greater than
twlnl rA,SIMM	**tw 5,**rA,SIMM	Trap if logically not less than
twlnli rA,SIMM	**twi 5,**rA,SIMM	Trap immediate if logically not less than
twlt rA,SIMM	**tw 16,**rA,SIMM	Trap if less than
twlti rA,SIMM	**twi 16,**rA,SIMM	Trap immediate if less than
twne rA,SIMM	**tw 24,**rA,SIMM	Trap if not equal
twnei rA,SIMM	**twi 24,**rA,SIMM	Trap immediate if not equal
twng rA,SIMM	**tw 20,**rA,SIMM	Trap if not greater than
twngi rA,SIMM	**twi 20,**rA,SIMM	Trap immediate if not greater than
twnl rA,SIMM	**tw 12,**rA,SIMM	Trap if not less than
twnli rA,SIMM	**twi 12,**rA,SIMM	Trap immediate if not less than

1 Simplified mnemonics for branch instructions that do not test a CR bit should not specify one; a programming error may occur.

2 The value in the BI operand selects CR*n*[2], the EQ bit.

3 Instructions for which B0 is either 12 (branch if condition true) or 4 (branch if condition false) do not depend on the CTR value and can be alternately coded by incorporating the condition specified by the BI field, as described in Section E.4.6, “Simplified Mnemonics that Incorporate CR Conditions (Eliminates BO and Replaces BI with crS).”

4 The value in the BI operand selects CR*n*[0], the LT bit.

5 The value in the BI operand selects CR*n*[1], the GT bit.

6 The value in the BI operand selects CR*n*[3], the SO bit.

Appendix F
Revision History

This appendix provides a list of the major differences between the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, Revision 0 through the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, Revision 3.

F.1 Changes From Revision 2 to Revision 3

Major changes to the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, from Revision 2 to Revision 3 are as follows:

Section, Page	Changes
Throughout	Referenced and changed bits are now called reference and change bits, to be consistent with the architecture specification.
	Code and pseudocode examples use colons (:) instead of en dashes (–) to indicate range. In regular text, en dashes are used to show range.
	The words 'interrupt' and 'exception' have been changed to match the architecture. The word storage is now used interchangeably with the word 'memory'.
	ISI and DSI exceptions are now called instruction storage interrupts and data storage interrupts.
	References to the 64-bit architecture have been largely removed.
	The term 'segment descriptor' has been replaced with specific references to 'segment registers'.
	Deleted all text pertaining to POWER compatibility, including Appendix B, “POWER Architecture Cross-References.”
4.4.3.1,4-63	In Table 4-37, added the following two sentences to the end of the **dcbi** operation:
	Note that some implementations may execute this instruction as a **dcbf**. This instruction is optional.
7.1,7-1	Replaced the fourth paragraph with the following:
	The segment information, used to generate the interim virtual addresses, is stored as segment registers.
7.3.1.1, 7-3	Removed this section.

8.2, 8-24 — Replaced the target address CTR || 0b00 of **bcctr** with CTR[0–29] || 0b00, the code sequence should read as follows:

```
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if cond_ok then
NIA ←iea CTR[0:29] || 0b00
if LK then LR ←iea CIA + 4
```

8.2, 8-25 — In the first sentence after Table 8-9, replaced the target address CTR || 0b00 of **bcctr** with CTR[0–29] || 0b00. The sentence should read as follows:

The branch target address is CTR[0–29] || 0b00.

8.2, 8-26 — Replaced the target address LR || 0b00 of **bclr** with CTR[0–29] || 0b00, the code sequence should read as follows:

```
if ¬ BO[2] then CTR ← CTR – 1
ctr_ok ← BO[2] | ((CTR ≠ 0) ⊕ BO[3])
cond_ok ← BO[0] | (CR[BI] ≡ BO[1])
if ctr_ok & cond_ok then
NIA ←iea LR[0:29] || 0b00
if LK then LR ←iea CIA + 4
```

8.2, 8-25 — In the first sentence after Table 8-11, replaced the target address LR || 0b00 of **bclr** with LR[0–29] || 0b00. The sentence should read as follows:

The branch target address is LR[0–29] || 0b00.

8.2, 8-43 — Added the following two sentences to the end of the **dcbi** description:

Note that some implementations may execute this instruction as a **dcbf**. This instruction is optional.

8.2, 8-44 — After the third paragraph, added the following paragraph:

The coherency state of a cache block after a write access (caused by a **dcbst**) is implementation-dependent. For example, some implementations may mark the cache block exclusive, where others may mark it invalid.

8.2, 8-163 — Replaced the first sentence of the only paragraph, the sentence should read as follows:

The contents of **r**S are shifted right the number of bits specified by **r**B[27–31].

Appendix A — Removed Section A-5, “Instructions Grouped by Functional Categories,” and Section A-6, “Instructions Sorted by Form.”

F.2 Changes From Revision 1 to Revision 2

Major changes to the *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture*, from Revision 1 to Revision 2 are as follows:

Section, Page	Change
2.1.1, 2-4	Figure 2-2 should show the GPR bit numbering as 0–31 (instead of 0–63).
2.1.6, 2-12	Figure 2-2 should show the link register bit numbering as 0–31 (instead of 0–63).
2.1.7, 2-12	Figure 2-2 should show the count register bit numbering as 0–31 (instead of 0–63).

3.1.3.1, 3-4	The first sentence in the last paragraph has a faulty cross reference. It should read as follows: The structure mapping introduces padding (skipped bytes indicated by (x) in Figure 3-2)...
3.1.6, 5-12	The section “Shared Memory,” has been added to clarify matters concerning memory access ordering.
7.4.6, 7-32	In Figure 7-11, PA0–PA63 should be PA00–PA31 for 32-bit implementations.
7.6.1, 7-49	In Figure 7-18, the size of a PTE should be 8 bytes not 16.
7.5.5, 7-47	Figure 7-16 should read “PA0–PA31<- RPN||A20–A31” instead of “PA0–PA31<- RPN||A30–A31.”
7.6.1.1, 7-51	Replaced the paragraph under the heading “Example” with the following: For example, suppose that the page table is 16,384 (2^{14}) 64-byte PTEGs, for a total size of 2^{20} bytes (1 Mbyte). A 14-bit index is required. Ten bits are provided from the hash to start with, so 4 additional bits from the hash must be selected. Thus the value in HTABMASK must be 15 and the value in HTABORG must have its low-order 4 bits (SDR1[12–15]) equal to 0. This means that the page table must begin on a $2^{<4 + 10 + 6>} = 2^{20}$ = 1-Mbyte boundary.
7.6.1.3, 7-53	Figure 7-20 should show that the primary hash is formed by XORing with page index bits 24–39 from the virtual address or EA[4–19] (instead of EA[24–39].
7.6.1.5, 7-56	The first sentence in the second paragraph should read as follows: “The physical address of the PTE(s) to be checked is derived as shown in Figure 7-21 and Figure 7-22, and the generated address is the address of a group of eight PTEs (a PTEG).”
7.6.3, 7-65	Deleted the third from the last paragraph in this section (beginning “Explicitly altering certain MSR bits...”)
8.2, 8-30	The eighth line of the code sequences should read “`CR[(4 * crfD) through (4 * crfD + 3)] ← c || XER[SO]`” instead of “`CR[4 * crfD-4 * crfD + 3] ← c || XER[SO].`”
8.2, 8-30	The fifth line of the code sequences should read “`CR[(4 * crfD) through (4 * crfD + 3)] ← c || XER[SO]`” instead of “`CR[4 * crfD-4 * crfD + 3] ← c || XER[SO].`”
8.2, 8-30	Added the following sentence at the end of the first paragraph: “The L bit has no effect on 32-bit operations.”
8.2, 8-33	The second to last sentence of the **cmpli** PowerPC instruction should read as follows: **cmpldi r**A**,** value equivalent to **cmpli 0,1,r**A**,**value
8.2, 8-132	The code sequence should read “`CR[(4 * crfD) through (4 * crfD + 3)] ← CR[(4 * crfS) through (4 * crfS + 3)]`” instead of “`CR[4 * crfD-4 * crfD + 3] ← CR[4 * crfS-4 * crfS + 3].`”

8.2, 8-136	Replaced the “low-order bits” with “low-order 32-bits” and “high-order bits” with “high-order 32-bits.”
8.2, 8-169	The first sentence should read “The contents of **r**S[0–31] are rotated left...” rather than “The contents of **r**S[32-63] are rotated left ...”
8.2, 8-173	The third line of the instruction description (if **r**B[58] = 0 then) should be removed.
8.2, 8-174	The second sentence in the second paragraph should read “The setting... , is independent of 32/64-bit mode” rather than “The setting..., is independent of mode.”
8.2, 8-175	The second sentence in the second paragraph should read “The setting.. , is independent of 32/64-bit mode” rather than “The setting..., is independent of mode.”
8.2, 8-213	The bit layout of **tlbie**[21–30] should read “306” rather than “30k6.”
A.3, A-14	In Table A-3, replaced “**subficx**” with“**subfic**.”
A.5, A-41	In Table A-42, replaced “**tlbiax**” with “**tlbia**” and “**tlbiex**” with “**tlbie**.”
A.5, A-41	In the column labelled “Optional” in Table A-42, instruction “**tlbsync**” should be flagged.

Glossary of Terms and Abbreviations

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from *IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

Note that some terms are defined in the context of how they are used in this book.

A

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details of how the processor or computer system must be implemented; instead it provides a template for a family of compatible *implementations*.

Asynchronous interrupt. *Interrupts* that are caused by events external to the processor's execution.

Atomic access. A bus access that attempts to be part of a read-write operation to the same address uninterrupted by any other access to that address (the term refers to the fact that the transactions are indivisible). The PowerPC architecture implements atomic accesses through the **lwarx**/**stwcx.** instruction pair.

B

BAT (block address translation) mechanism. A software-controlled array that stores the available block address translations on-chip.

Biased exponent. An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address n of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 being the most-significant byte. *See* Little-endian.

Block. An area of memory that ranges from 128 Kbyte to 256 Mbyte, whose size, translation, and protection attributes are controlled by the *BAT mechanism*.

Boundedly undefined. A characteristic of results of certain operations that are not rigidly prescribed by the PowerPC architecture. Boundedly- undefined results for a given operation may vary among implementations, and between execution attempts in the same implementation.

Although the architecture does not prescribe the exact behavior for when results are allowed to be boundedly undefined, the results of executing instructions in contexts where results are allowed to be boundedly undefined are constrained to ones that could have been achieved by executing an arbitrary sequence of defined instructions, in valid form, starting in the state the machine was in before attempting to execute the given instruction.

C

Cache High-speed memory component containing recently-accessed data and/or instructions (subset of main memory).

Cache block. A small region of contiguous memory that is copied from memory into a *cache*. The size of a cache block may vary among processors; the maximum block size is one *page*. In PowerPC processors, *cache coherency* is maintained on a cache-block basis. Note that the term ‘cache block’ is often used interchangeably with ‘cache line’.

Cache coherency. An attribute wherein an accurate and common view of memory is provided to all devices that share the same memory system. Caches are coherent if a processor performing a read from its cache is supplied with data corresponding to the most recent value written to memory or to another processor’s cache.

Cache flush. An operation that removes from a cache any data from a specified address range. This operation ensures that any modified data within the specified address range is written back to main memory. This operation is generated typically by a Data Cache Block Flush (**dcbf**) instruction.

Caching-inhibited. A memory update policy in which the *cache* is bypassed and the load or store is performed to or from main memory.

Cast-outs. *Cache blocks* that must be written to memory when a cache miss causes a cache block to be replaced.

Change bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the change bit if any store is performed into the *page*. *See also* Page access history bits and Reference bit.

Clear. To cause a bit or bit field to register a value of zero. *See also* Set.

Context synchronization. An operation that ensures that all instructions in execution complete past the point where they can produce an *interrupt*, that all instructions in execution complete in the context in which they began execution, and that all subsequent instructions are *fetched* and executed in the new context. Context synchronization may result from executing specific instructions (such as **isync** or **rfi**) or when certain events occur (such as an interrupt).

Copy-back. An operation in which modified data in a *cache block* is copied back to memory.

D

Denormalized number. A nonzero floating-point number whose *exponent* has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

Direct-mapped cache. A cache in which each main memory address can appear in only one location within the cache, operates more quickly when the memory request is a cache hit.

Direct-store. Interface available on PowerPC processors only to support direct-store devices from the POWER architecture. When the T bit of a *segment descriptor* is set, the descriptor defines the region of memory that is to be used as a direct-store segment. Note that this facility is being phased out of the architecture and will not likely be supported in future devices. Therefore, software should not depend on it and new software should not use it.

E

Effective address (EA). The 32-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exception. A condition encountered by the processor that, if enabled, causes an *interrupt*.

Interrupt handler. A software routine that executes when an interrupt is taken. Normally, the interrupt handler corrects the condition that caused the interrupt, or performs some other meaningful task (that may include aborting the program that caused the interrupt). The address for each interrupt handler is identified by an interrupt vector offset defined by the architecture and a prefix specified by the MSR[IP].

Extended opcode. A secondary opcode field generally located in instruction bits 21–30, that further defines the instruction type. All PowerPC instructions are one word in length. The most significant 6 bits of the instruction are the *primary opcode*, identifying the type of instruction. *See also* Primary opcode.

Execution synchronization. A mechanism by which all instructions in execution are architecturally complete before beginning execution (appearing to begin execution) of the next instruction. Similar to context synchronization but doesn't force the contents of the instruction buffers to be deleted and refetched.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. *See also* Biased exponent.

F

Fetch. Retrieving instructions from either the cache or main memory and placing them into the instruction queue.

Floating-point register (FPR). Any of the 32 registers in the floating-point register file. These registers provide the source operands and destination results for floating-point instructions. Load instructions move data from memory to FPRs and store instructions move data from FPRs to memory. The FPRs are 64 bits wide and store floating-point values in double-precision format.

Fraction. In the binary representation of a floating-point number, the field of the *significand* that lies to the right of its implied binary point.

Fully-associative. Addressing scheme where every cache location (every byte) can have any possible address.

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

Guarded. The guarded attribute pertains to speculative (out-of-order) execution. When a page is designated as guarded, instructions and data cannot be accessed speculatively.

H

Harvard architecture. An architectural model featuring separate cache and memory management resources for instruction and data.

Hashing. An algorithm used in the *page table* search process.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

Illegal instructions. A class of instructions that are not implemented for a particular PowerPC processor. These include instructions not defined by the PowerPC architecture. In addition, for 32-bit implementations, instructions that are defined only for 64-bit implementations are considered to be illegal instructions.

Implementation. A particular processor that conforms to the PowerPC architecture, but may differ from other architecture-compliant implementations for example in design, feature set, and implementation of *optional* features. The PowerPC architecture has many different implementations.

Implementation-dependent. An aspect of a feature in a processor's design that is defined by a processor's design specifications rather than by the PowerPC architecture. When functionality is identified as implementation-dependent refer to the implementation's reference manual for device specific details.

Implementation-specific. An aspect of a feature in a processor's design that is not required by the PowerPC architecture, but for which the PowerPC architecture may provide concessions to ensure that processors that implement the feature do so consistently.

Imprecise interrupt. A type of *synchronous interrupt* that is allowed not to adhere to the precise interrupt model (*see* Precise interrupt). The PowerPC architecture allows only floating-point interrupts to be handled imprecisely.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

In-order. *See* nonspeculative.

Instruction latency. The total number of clock cycles necessary to execute an instruction and make ready the results of that instruction.

Instruction parallelism. A feature of PowerPC processors that allows instructions to be processed in parallel.

Interrupt. An event caused by an *exception* that requires special, supervisor-level processing.

Invalid state. State of a cache entry that does not currently contain a valid copy of a cache block from memory.

K

Key bits. A set of key bits referred to as Ks and Kp in each segment register and each BAT register. The key bits determine whether supervisor or user programs can access a *page* within that *segment* or *block.*

Kill. An operation that causes a *cache block* to be invalidated.

L

L2 cache. *See* Secondary cache.

Least-significant bit (lsb). The bit of least value in an address, register, data element, or instruction encoding.

Least-significant byte (LSB). The byte of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 being the *most-significant byte*. *See* Big-endian.

M

MESI (modified/exclusive/shared/invalid). *Cache coherency* protocol used to manage caches on different devices that share a memory system. Note that the PowerPC

architecture does not specify the implementation of a MESI protocol to ensure cache coherency.

Memory access ordering. The specific order in which the processor performs load and store memory accesses and the order in which those accesses complete.

Memory-mapped accesses. Accesses whose addresses use the page or block address translation mechanisms provided by the MMU and that occur externally with the bus protocol defined for memory.

Memory coherency. An aspect of caching in which it is ensured that an accurate view of memory is provided to all devices that share system memory.

Memory consistency. Refers to agreement of levels of memory with respect to a single processor and system memory (for example, on-chip cache, secondary cache, and system memory).

Memory management unit (MMU). The functional unit that is capable of translating an *effective* (logical) *address* to a physical address, providing protection mechanisms, and defining caching methods.

Microarchitecture. The hardware details of a microprocessor's design. Such details are not defined by the PowerPC architecture.

Mnemonic. The abbreviated name of an instruction used for coding.

Modified state. When a cache block is in the modified state, it has been modified by the processor since it was copied from memory. *See* MESI.

Munging. A modification performed on an *effective address* that allows it to appear to the processor that individual aligned scalars are stored as *little-endian* values, when in fact it is stored in *big-endian* order, but at different byte addresses within double words. Note that munging affects only the effective address and not the byte order. Note also that this term is not used by the PowerPC architecture.

Multiprocessing. The capability of software, especially operating systems, to support execution on more than one processor at the same time.

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

Most-significant byte (MSB). The highest-order byte in an address, registers, data element, or instruction encoding.

N

NaN . An abbreviation for 'Not a Number'; a symbolic entity encoded in floating-point format. There are two types of NaNs—signaling NaNs (SNaNs) and quiet NaNs (QNaNs).

Nonspeculative. An aspect of an operation that adheres to a sequential model. An operation is said to be nonspeculative if, at the time that it is performed, it is known to be required by the sequential execution model. *See* speculative.

No-op. No-operation. A single-cycle operation that does not affect registers or generate bus activity.

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

OEA (operating environment architecture). The level of the architecture that describes PowerPC memory management model, supervisor-level registers, synchronization requirements, and the interrupt model. It also defines the time-base feature from a supervisor-level perspective. Implementations that conform to the PowerPC OEA also conform to the PowerPC UISA and VEA.

Optional. A feature, such as an instruction, a register, or an interrupt, that is defined by the PowerPC architecture but not required to be implemented.

Out-of-order memory access. A memory access performed ahead of one that may have preceded it in the sequential model, such as is allowed by a weakly-ordered memory model. *See* speculative.

Out-of-order execution. A technique that allows instructions to be issued and completed in an order that differs from their sequence in the instruction stream.

Overflow. An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits.

P

Page. A unit of memory for which address translation is independently specifiable.

Page access history bits. The *change* and *reference* bits in the PTE keep track of the access history within the page. The reference bit is set by the MMU whenever the page is accessed for a read or write operation. The change bit is set when the page is stored into. *See* Change bit and Reference bit.

Page fault. A page fault is a condition that occurs when the processor attempts to access a memory location that resides within a *page* not currently resident in *physical memory*. On PowerPC processors, a page fault interrupt condition occurs when a matching, valid *page table entry* (PTE[V] = 1) cannot be located.

Page table. A table in memory is comprised of *page table entries*, or PTEs. It is further organized into eight PTEs per PTEG (page table entry group). The number of PTEGs in the page table depends on the size of the page table (as specified in the SDR1 register).

Page table entry (PTE). Data structures containing information used to translate *effective address* to physical address on a 4-Kbyte page basis. A PTE consists of 8 bytes of information in a 32-bit processor and 16 bytes of information in a 64-bit processor.

Physical memory. The actual memory that can be accessed through the system's memory bus.

Pipelining. A technique that breaks operations, such as instruction processing or bus transactions, into smaller distinct stages or tenures (respectively) so that a subsequent operation can begin before the previous one has completed.

Precise interrupts. A category of interrupt for which the pipeline can be stopped so instructions that preceded the faulting instruction can complete, and subsequent instructions can be flushed and redispatched after interrupt handling has completed. *See* Imprecise interrupts.

Primary opcode. The most-significant 6 bits (bits 0–5) of the instruction encoding that identifies the type of instruction. S*ee* Secondary opcode.

Protection boundary. A boundary between *protection domains*.

Protection domain. A protection domain is a segment, a virtual page, a BAT area, or a range of unmapped effective addresses. It is defined only when the appropriate relocate bit in the MSR (IR or DR) is 1.

Q

Quad word. A group of 16 contiguous locations starting at an address divisible by 16.

Quiet NaN. A type of *NaN* that can propagate through most arithmetic operations without signaling exceptions. A quiet NaN is used to represent the results of certain invalid operations, such as invalid arithmetic operations on infinities or on NaNs, when invalid. *See* Signaling NaN.

R

rA. The **r**A instruction field is used to specify a GPR to be used as a source or destination.

rB. The **r**B instruction field is used to specify a GPR to be used as a source.

rD. The **r**D instruction field is used to specify a GPR to be used as a destination.

rS. The **r**S instruction field is used to specify a GPR to be used as a source.

Real address mode. An MMU mode when no address translation is performed and the *effective address* specified is the same as the physical address. The processor's MMU is operating in real address mode if its ability to perform address translation has been disabled through the MSR registers IR and/or DR bits.

Record bit. Bit 31 (or the Rc bit) in the instruction encoding. When it is set, updates the condition register (CR) to reflect the result of the operation.

Reference bit. One of two *page history bits* found in each *page table entry* (PTE). The processor sets the *reference bit* whenever the page is accessed for a read or write. *See also* Page access history bits.

Register indirect addressing. A form of addressing that specifies one GPR that contains the address for the load or store.

Register indirect with immediate index addressing. A form of addressing that specifies an immediate value to be added to the contents of a specified GPR to form the target address for the load or store.

Register indirect with index addressing. A form of addressing that specifies that the contents of two GPRs be added together to yield the target address for the load or store.

Reservation. The processor establishes a reservation on a *cache block* of memory space when it executes an **lwarx** instruction to read a memory semaphore into a GPR.

Reserved field. In a register, a reserved field is one that is not assigned a function. A reserved field may be a single bit. The handling of reserved bits is *implementation-dependent*. Software is permitted to write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

RISC (reduced instruction set computing). An *architecture* characterized by fixed-length instructions with nonoverlapping functionality and by a separate set of load and store instructions that perform memory accesses.

S

Scalability. The capability of an architecture to generate *implementations* specific for a wide range of purposes, and in particular implementations of significantly greater performance and/or functionality than at present, while maintaining compatibility with current implementations.

Secondary cache. A cache memory that is typically larger and has a longer access time than the primary cache. A secondary cache may be shared by multiple devices. Also referred to as L2, or level-2, cache.

Segment. A 256-Mbyte area of *virtual memory* that is the most basic memory space defined by the PowerPC architecture. Each segment is configured through a unique *segment descriptor.*

Segment descriptors. Information used to generate the interim *virtual address*. The segment descriptors reside in 16 on-chip segment registers for 32-bit implementations. For 64-bit implementations, the segment descriptors reside as *segment table entries* in a hashed segment table in memory.

Set (*v*). To write a nonzero value to a bit or bit field; the opposite of *clear*. The term ‘set’ may also be used to generally describe the updating of a bit or bit field.

Set (*n*). A subdivision of a *cache*. Cacheable data can be stored in a given location in any one of the sets, typically corresponding to its lower-order address bits. Because several memory locations can map to the same location, cached data is typically placed in the set whose *cache block* corresponding to that address was used least recently. *See* Set-associative.

Set-associative. Aspect of cache organization in which the cache space is divided into sections, called *sets*. The cache controller associates a particular main memory address with the contents of a particular set, or region, within the cache.

Signaling NaN. A type of *NaN* that generates an invalid operation program interrupt when it is specified as arithmetic operands. *See* Quiet NaN.

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

Speculative memory access. An access to memory that occurs before it is known to be required by the sequential execution model.

Simplified mnemonics. Assembler mnemonics that represent a more complex form of a common operation.

Static branch prediction. Mechanism by which software (for example, compilers) can give a hint to the machine hardware about the direction a branch is likely to take.

Sticky bit. A bit that when *set* must be cleared explicitly.

Strong ordering. A memory access model that requires exclusive access to an address before making an update, to prevent another device from using stale data.

Superscalar machine. A machine that can issue multiple instructions concurrently from a conventional linear instruction stream.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and can access the supervisor memory space, among other privileged operations.

Synchronization. A process to ensure that operations occur strictly *in order*. *See* Context synchronization and Execution synchronization.

Synchronous interrupt. An *interrupt* that is generated by the execution of a particular instruction or instruction sequence. There are two types of synchronous interrupts, *precise* and *imprecise*.

System memory. The physical memory available to a processor.

T

TLB (translation lookaside buffer) A cache that holds recently-used *page table entries*.

Throughput. The measure of the number of instructions that are processed per clock cycle.

Tiny. A floating-point value that is too small to be represented for a particular precision format, including *denormalized* numbers; they do not include ±0.

U

UISA (user instruction set architecture). The level of the architecture to which user-level software should conform. The UISA defines the base user-level instruction set, user-level registers, data types, floating-point memory conventions and interrupt model as seen by user programs, and the memory and programming models.

Underflow. An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or mantissa than the single-precision format can provide. In other words, the result is too small to be represented accurately.

Unified cache. Combined data and instruction cache.

User mode. The unprivileged operating state of a processor used typically by application software. In user mode, software can only access certain control registers and can access only user memory space. No privileged operations can be performed. Also referred to as problem state.

V

VEA (virtual environment architecture). The level of the *architecture* that describes the memory model for an environment in which multiple devices can access memory, defines aspects of the cache model, defines cache control instructions, and defines the time-base facility from a user-level perspective. *Implementations* that conform to the PowerPC VEA also adhere to the UISA, but may not necessarily adhere to the OEA.

Virtual address. An intermediate address used in the translation of an *effective address* to a physical address.

Virtual memory. The address space created using the memory management facilities of the processor. Program access to virtual memory is possible only when it coincides with *physical memory.*

W

Weak ordering. A memory access model that allows bus operations to be reordered dynamically, which improves overall performance and in particular reduces the effect of memory latency on instruction throughput.

Word. A 32-bit data element.

Write-back. A cache memory update policy in which processor write cycles are directly written only to the cache. External memory is updated only indirectly, for example, when a modified cache block is *cast out* to make room for newer data.

Write-through. A cache memory update policy in which all processor write cycles are written to both the cache and memory.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z